工业和信息化部高等教育“十三五”规划教材

C语言程序设计

主　编　袁　敏　崔　健
副主编　逄锦聚　王兆堃　代令军
　　魏　东　于韶杰　胡凤珠

電子工業出版社
Publishing House of Electronics Industry
北京・BEIJING

图书在版编目(CIP)数据

C 语言程序设计 / 袁敏, 崔健主编. —北京: 电子工业出版社, 2017.8
ISBN 978-7-121-32549-6

Ⅰ. ①C… Ⅱ. ①袁… ②崔… Ⅲ. ①C 语言－程序设计 Ⅳ. ①TP312.8

中国版本图书馆 CIP 数据核字(2017)第 203175 号

策划编辑：郝国栋
责任编辑：郝国栋
印　　刷：
装　　订：
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×1092　1/16　　印张：15.5　　字数：360 千字
版　　次：2017 年 8 月第 1 版
印　　次：2017 年 8 月第 1 次印刷
定　　价：30.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：（0532）67772605，邮箱：majie@phei.com.cn

目 录

第 1 章 C 语言程序设计概述 …… 1
1.1 程序与程序设计语言 …… 1
1.1.1 程序与程序设计 …… 1
1.1.2 程序设计语言 …… 2
1.1.3 算法与数据结构 …… 3
1.2 C 语言概述 …… 3
1.2.1 C 语言的特点 …… 3
1.2.2 C 语言程序的基本结构 …… 4
1.2.3 C 语言的基本要素 …… 5
1.3 设计 C 语言程序的基本过程 …… 6
1.3.1 C 语言程序编程环境 …… 6
1.3.2 编译、连接和运行 …… 8
1.4 如何学习 C 语言 …… 8
习题 1 …… 9
第 2 章 数据类型和表达式 …… 11
2.1 C 语言数据类型 …… 12
2.2 变量 …… 13
2.3 常量 …… 15
2.3.1 整型常量 …… 15
2.3.2 实型常量 …… 16
2.3.3 字符型常量 …… 16
2.3.4 符号常量 …… 17
2.4 数据的输入、输出 …… 18
2.4.1 printf()函数 …… 18
2.4.2 scanf()函数 …… 20
2.5 运算符与表达式 …… 23
2.5.1 算术运算符 …… 23
2.5.2 关系运算符 …… 25
2.5.3 逻辑运算符 …… 25
2.5.4 赋值运算符 …… 26
2.5.5 条件运算符 …… 27
2.5.6 逗号运算符 …… 27
2.5.7 位运算符 …… 28

2.6 类型转换 …… 29
2.6.1 自动类型转换 …… 29
2.6.2 强制类型转换 …… 30
习题 2 …… 31
第 3 章 程序控制结构 …… 33
3.1 概述 …… 33
3.2 顺序结构 …… 34
3.3 选择结构 …… 36
3.3.1 单分支结构 …… 36
3.3.2 双分支结构 …… 37
3.3.3 分支结构的嵌套 …… 38
3.3.4 多路分支结构 …… 41
3.4 循环结构 …… 46
3.4.1 for 循环结构 …… 47
3.4.2 while 循环结构 …… 49
3.4.3 do-while 循环结构 …… 50
3.4.4 循环结构的嵌套 …… 52
3.5 break 和 continue 控制语句 …… 53
3.5.1 break 语句 …… 53
3.5.2 continue 语句 …… 54
3.6 程序控制结构的综合应用 …… 56
习题 3 …… 61
第 4 章 数组 …… 64
4.1 一维数组 …… 64
4.1.1 一维数组的定义和引用 …… 64
4.1.2 一维数组的初始化 …… 66
4.1.3 一维数组编程实例 …… 66
4.2 一维字符数组和字符串 …… 74
4.2.1 一维字符数组的定义和初始化 …… 74
4.2.2 字符串概念 …… 75
4.2.3 字符串存储 …… 75
4.2.4 字符串输出 …… 77
4.2.5 字符串的处理 …… 78
4.3 二维数组 …… 81
4.3.1 二维数组的定义和引用 …… 81
4.3.2 二维数组的初始化 …… 82
4.3.3 二维数组编程实例 …… 83
习题 4 …… 87
第 5 章 函数 …… 88
5.1 模块化程序设计 …… 88
5.2 函数的定义和调用 …… 90
5.2.1 函数的定义 …… 90
5.2.2 函数的调用 …… 92

5.3 变量的存储属性 …… 97
5.3.1 自动(auto)变量 …… 98
5.3.2 寄存器(register)变量 …… 99
5.3.3 静态(static)变量 …… 99
5.3.4 用 extern 声明外部变量 …… 101
5.4 函数的嵌套调用 …… 104
5.5 递归函数 …… 105
5.6 数组作函数参数 …… 110
5.6.1 数组元素作函数实参 …… 110
5.6.2 一维数组名作函数参数 …… 111
5.6.3 二维数组名作函数参数 …… 114
习题 5 …… 117
第 6 章 指针 …… 118
6.1 指针的概念 …… 118
6.2 指针与简单变量 …… 120
6.2.1 指针变量的定义与引用 …… 120
6.2.2 指针变量的特殊性 …… 122
6.2.3 指针变量作为函数的参数 …… 123
6.3 指针与一维数组 …… 127
6.3.1 数组名是一个指针常量 …… 127
6.3.2 指针的运算 …… 129
6.3.3 将数组地址传递给函数 …… 131
6.4 指针与字符串 …… 135
6.4.1 使用指针表示字符串 …… 135
6.4.2 动态内存分配 …… 136
6.4.3 常用的字符串处理函数 …… 138
6.5 指针进阶 …… 144
6.5.1 二级指针 …… 144
6.5.2 指针与二维数组 …… 145
6.5.3 指针数组 …… 145
6.5.4 命令行参数 …… 149
6.5.5 返回指针的函数 …… 151
6.5.6 指向函数的指针 …… 152
习题 6 …… 154
第 7 章 构造数据类型与预编译处理 …… 156
7.1 结构体 …… 156
7.1.1 结构体类型的定义 …… 157
7.1.2 结构体变量的定义 …… 158
7.1.3 结构体变量的引用 …… 161
7.1.4 结构体变量的初始化 …… 161
7.2 结构体数组 …… 163
7.3 线性链表 …… 167
7.3.1 链表的概念 …… 168

7.3.2 链表的基本操作 …… 169
7.4 共用体 …… 174
7.4.1 共用体类型定义 …… 174
7.4.2 共用体变量的定义、引用 …… 175
7.4.3 共用体变量的赋值 …… 175
7.5 枚举类型 …… 177
7.6 自定义类型名 …… 179
7.7 编译预处理 …… 180
7.7.1 编译预处理命令简介 …… 180
7.7.2 宏定义 …… 181
7.7.3 文件包含 …… 185
7.7.4 条件编译 …… 186
习题 7 …… 189
第 8 章 文件 …… 192
8.1 文件的基本概念 …… 192
8.2 文件指针 …… 193
8.3 文件的打开、读写与关闭 …… 194
8.3.1 文件的打开 …… 194
8.3.2 文件的关闭 …… 195
8.3.3 文件的读写 …… 195
8.3.4 文件读写函数的选用原则 …… 203
8.4 文件定位 …… 203
习题 8 …… 205
附录 …… 208
附录Ⅰ ASCII 码表 …… 208
附录Ⅱ C 标准库函数 …… 209
2.1 输入与输出函数<stdio.h> …… 209
2.2 字符类测试函数<ctype.h> …… 216
2.3 字符串函数<string.h> …… 217
2.4 数学函数<math.h> …… 220
2.5 实用函数<stdlib.h> …… 222
2.6 诊断函数<assert.h> …… 224
2.7 变长变元表函数<stdarg.h> …… 225
2.8 非局部跳转函数<setjmp.h> …… 225
2.9 信号处理函数<signal.h> …… 226
2.10 日期与时间函数<time.h> …… 226
2.11 由实现定义的限制<limits.h>和<float.h> …… 228
附录Ⅲ C 语言错误提示 …… 230
3.1 致命错误信息 …… 230
3.2 一般错误信息 …… 230
附录Ⅳ 编程风格 …… 236
附录Ⅴ 全国计算机等级考试二级C语言程序设计考试大纲(2013年版) … 238

第 1 章　C 语言程序设计概述

C 语言是面向过程的高级程序设计语言之一，它具有数据类型丰富、灵活高效和结构化等特征。本章主要介绍 C 语言的来历，C 语言的特征和如何使用 VC6.0 设计一个简单的 C 程序，并编译和运行这个 C 程序，最后就如何学好 C 语言给出建议。

知 识 结 构

1. 程序与程序设计语言
   ① 程序
   ② 程序设计
   ③ 算法与数据结构
   ④ 程序设计语言的发展
2. C 语言概述
   ① C 语言的特点
   ② C 语言程序的基本结构
   ③ C 语言的基本要素
3. C 程序设计基本过程
   ① C 编程环境
   ② 编辑源程序
   ③ 编译、连接、运行
4. 如何学好 C 语言

1.1　程序与程序设计语言

1.1.1　程序与程序设计

程序是解决特定问题的步骤，计算机程序是用计算机程序设计语言编写的，最终能够在计算机上运行的指令的序列。计算机执行的每一个操作，都是按预先设好的指令完成的。如图 1.1 所示，勇气号火星探测器在火星的行走、对岩石取样等都是一个个具体的任务，都需要设好指令。用指令编写成能够完成一个具体任务的序列，这就是程序。

程序设计是指设计、编制、调试程序的方法和过程。程序设计通常分为问题建模、算法设计、编写代码和编译调试几个阶段。

图 1.1　勇气号火星探测器

1.1.2　程序设计语言

语言作为一种交流的工具，其要素有两点：一组符号和一组规则，按照规则由记号构成的记号串的总体就是语言。程序设计语言，简称编程语言，就是人与计算机交流的工具，用来向计算机发出指令，使得计算机能够准确完成解决特定问题设定的步骤。在程序设计语言中，记号串就是程序。程序设计语言的基本成分有：

① 数据成分—— 描述程序处理的对象，在计算机中统称为数据。

② 运算成分—— 描述程序中对数据的处理，即运算。

③ 控制成分—— 描述程序的流程控制。

④ 传输成分—— 描述对数据的传输，即输入和输出。

计算机程序设计语言的发展，经历了从机器语言、汇编语言到高级语言的历程，目前面向对象的程序设计思想和软件开发技术已经成为主流。

1. 机器语言

机器语言是指由二进制编码表示的一种语言。从始至今，计算机内部能识别的只有 0 和 1 构成的二进制代码。最早的程序都是采取二进制的机器语言编写的，编程者要精通计算机的内部结构和熟记 0 和 1 构成的机器指令才能够设计程序，这相当困难，影响了计算机的普及和应用。

2. 汇编语言

为了解决计算机编程难的问题，人们在 1952 年研究了能够代替机器语言的助记符语言，即汇编语言。汇编语言通过助记符代替机器指令，克服了机器语言难记和不易读等缺点，但是汇编语言依赖于特定机器，可移植性差，一般只用在系统软件和特定领域的开发。

3. 面向过程的高级语言

为了普及计算机，需要编制各领域的应用软件，让计算机发挥更大的作用，人们需要更易于掌握的和接近人类自然语言的编程语言，面向过程的高级语言就产生了。计算机不能直接运行使用高级语言编写的程序，需要编译或解释系统把它“翻译”成机器语言程序才能运行。不同的高级语言有不同的编译系统。

高级语言种类很多，有 Fortran、Basic、ALGOL、C 等。面向过程的高级语言主要是面向数学公式和算法的语言。具有丰富的数据类型、结构化的语句、模块化的结构、简单易懂、高效等特点。

4. 面向对象的高级语言

随着编写大型应用软件的需要，20 世纪 80 年代，产生了更接近人类自然交流的语言，如：C++，Perl，Visual Basic，Java 等。计算机程序设计和软件开发全面进入面向对象的程序设计阶段。

随着各种软件系统的设计实现，计算机已经深入到国防安全、工业控制、金融通信、互联网等有关生活的方方面面。

1.1.3　算法与数据结构

著名的计算机科学家，Pascal 之父——Niklaus·Wirth 曾提出一个公式：

数据结构+算法=程序

他因此获得了计算机界的诺贝尔奖——图灵奖。

1. 算法

算法就是为解决特定问题所采取的方法和步骤，解决不同问题所采用的算法不同。

在程序设计中，算法就是一系列解决问题的清晰指令。如果一个算法有缺陷或者不适合解决某个问题，执行这个算法，问题将得不到解决。解决同一问题可能有不同的算法，效率可能大不一样，衡量算法的优劣指标有时间复杂度和空间复杂度。

算法的 5 大特征：

① 有穷性—— 一个算法必须保证执行有限步之后结束。

② 确切性—— 算法的每一步骤必须有确切的定义。

③ 可行性—— 原则上算法能够精确地运行。

④ 零个或多个输入—— 一个算法可以有零个或多个输入。

⑤ 一个或多个输出—— 一个算法必须有输出，没有输出的算法毫无意义。

2. 数据结构

数据结构即计算机存储、组织数据的方式。数据结构是指相互之间存在一种或几种特定关系的数据的集合。数据结构与算法密不可分，一个良好的数据结构可以简化算法，反过来，明确了问题的算法，才能较好地设计数据结构，两者相辅相成。通常情况下，精心选择的数据结构可以带来更高的运行效率和存储效率。

1.2　C 语言概述

1.2.1　C 语言的特点

C 语言是近些年来最流行的计算机高级语言，根据 TIOBE 的统计，C 语言的用户占有量在 Java 出现之前一直排在第一位。直到 2000 年 Java 出现后 C 语言才有了对手。如图 1.2 所示，2010 年 11 月以来 C 语言又多次回到第一的位置。这主要是因为 C 语言既可以用来编写系统软件，也可以用来编写应用软件。C 语言的主要特点如下。

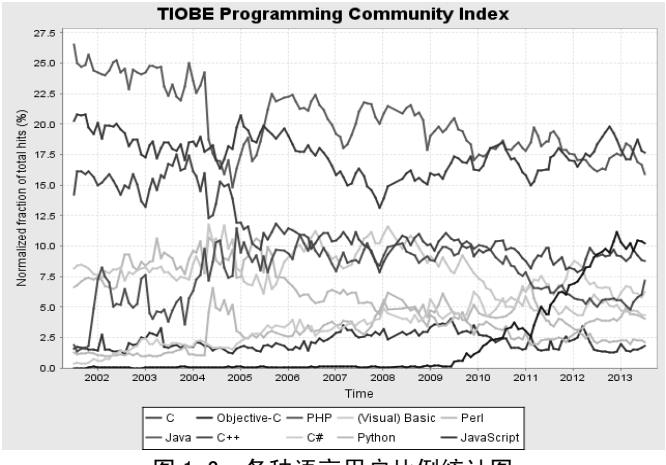


图 1.2　各种语言用户比例统计图

① 语言简洁、紧凑，使用方便灵活。C 语言一共有 32 个关键字，9 种控制语句，程序书写形式自由，语法限制少。

② 运算符丰富、数据处理能力强。C 语言的运算符包含的范围很广，共有 34 种运算符。

③ 数据结构丰富，具有现代高级语言提供的大多数数据结构。C 语言的数据类型有整型、实型、字符型、数组类型、指针类型、结构体类型等。C 语言能用来实现各种复杂的数据结构的运算，尤其是指针型数据，使用起来比较灵活、多样。

④ C 语言是一种结构化的程序设计语言。C 语言具有结构化的控制语句，并用函数作为程序的模块单位，是一种理想的结构化的编程语言。

⑤ C 语言允许直接访问物理地址，能进行位(bit)操作，实现汇编语言的大部分功能，还可以直接对硬件进行操作。

⑥ 可移植性好(与汇编语言相比)。源代码基本上不做修改，经过重新编译，就能够用于各种操作系统。

当然，C 语言也有其不足之处。例如，C 语言的语法限制不太严格，在增加程序设计的灵活性的同时，在一定程度上降低了某些安全性，这对程序设计人员提出了更高的要求。

由于开发大型软件的需要，近年来，面向对象的 C++语言、Java 语言得到进一步推广，但是不应该得出“C 语言已经过时了，不必学习 C 语言了”的结论。不应该把面向对象的程序设计与面向过程的程序设计对立起来，面向对象的基础是面向过程。自 20 世纪 90 年代以来，我国学习使用 C 语言的人越来越多，学习 C 语言不仅能培养我们的编程能力，同时还能使我们养成良好的逻辑思维习惯，培养我们分析问题和解决问题的能力。

1.2.2 C 语言程序的基本结构

【例 1.1】一个 C 语言程序示例。

```
/**********************/
/*    C 语言程序实例    */
/**********************/
#include<stdio.h>                    //编译预处理命令
void  swap(int  x, int  y)           //函数定义，函数首部
{                                    //函数体
   int  t;                           //定义变量
   t=x; x=y; y=t;
   printf("%d  %d", x, y);           //输出
}
main( )                              //主函数定义
{
   int  a=3, b=4;
   swap(a, b);
   printf("%d  %d", a, b);
   return  0;
}
```

由此，可以分析 C 语言程序的基本结构。

① 函数是 C 语言程序(简称 C 程序)的基本构成单位。

C 程序由一个或多个函数构成，函数是构成 C 程序的基本单位。一个 C 程序中必须包含一个主函数 main()，而且最多只能有一个主函数。除了一个主函数外还可以有零个或多个其

他函数，如 swap()函数。

② 一个 C 程序总是从 main()函数开始执行，至 main()函数的最后一句结束，函数中可以调用其他函数。

③ 函数的一般结构：

函数由函数首部加函数体构成，结构如下：

```
返回值类型 函数名(参数列表)
{
  函数体
}
```

返回值类型可以缺省，如 main()，缺省时表示返回整型值，等价于 int main()。参数列表可以没有，表示没有参数，也可以写成 main(void)，但函数名后的一对圆括号不能省略，这是函数的标志。函数体紧跟函数首部，以左大括号“{”开始，以右大括号“}”结束。通常包括变量说明部分和执行语句部分，说明部分必须在执行部分之前。

C 语言中函数分两种，自定义函数和库函数，库函数是 C 编译系统提供的，如标准输入输出函数 scanf()和 printf()等，它们的定义存储在“stdio.h”文件中。因此当要调用 printf()、scanf()等函数时需要在程序的开头加上一条编译预处理命令#include<stdio.h>。

④ 执行语句以分号结尾，分号是语句的一部分，表示一条语句的结束。

⑤ 注释部分是“/*”和“*/”括起来的部分，它对程序的功能或含义加以说明，编译时不起作用。注释部分可以出现在程序任何位置，单行注释也可以用“//”开头。

⑥ C 语言本身没有输入输出语句。由于输入输出操作涉及具体的计算机设备，因此 C 语言对输入输出实行“函数化”，把输入输出操作放在函数中处理。输入和输出的操作由库函数完成，如 scanf()和 printf()。

建议用 C 语言编程时，每行写一条语句，注意缩进、留空、对齐、加注释，养成良好的编程风格，使得程序结构清晰，可读性强，方便排错、调试、修改。

1.2.3　C 语言的基本要素

1. 关键字

关键字是 C 语言规定的具有特定意义的字符串。一共有 32 个关键字，由系统定义，如表 1.1 所示。关键字又称为保留字，它们不能重新用作其他定义。

表 1.1　C 语言的关键字

auto	break	case	char	const	continue	default	do
double	else	enum	extern	float	for	goto	if
int	long	register	return	short	signed	sizeof	static
struct	switch	typedef	unsigned	union	void	volatile	while

2. 标识符

简单地说，标识符就是一个名字。用来表示符号常量名、函数名、数组名、类型名、文件名的有效字符序列称为标识符(identifier)。程序设计语言中的标识符均有命名规则，C 语言中标识符有 3 类：关键字、预定义标识符、自定义标识符。

① 预定义标识符：C 语言编译系统提供的库函数名和编译预处理命令等构成了预定义标识符。

② 自定义标识符：用户可自定义标识符，C 语言标识符命名规则规定，标识符只能由字母、数字和下划线三种字符组成，且第一个字符必须是字母或下划线。

如下面列出的几个字串就是合法的标识符：

Sum	average	_total	class	day	Month
student_name	tan	lotus_1_2_3	basic	li_ling	

下面列出的几个字串不是合法的标识符：

M.	d.j	ohn，	y 123	＃33	3d64	a>b

C 语言区分大小写字母。因此，A 和 a，I 和 i，Sum 和 sum 分别表示两个不同的标识符。

标识符的命名应该做到见名知义，如 max，min，sum，count 等，另外要符合业内命名习惯，如变量名一般用小写字母，符号常量名用大写字母，i，j，k 通常用作循环控制变量名。

3. 库函数

C 语言的库函数是系统预先定义好的由编译系统提供的函数，包含了许多通用的常用的功能模块，库函数的引入，极大方便了程序设计人员的使用，也扩大了语言本身的功能。

库函数按照功能分为若干类，程序中如果需要用到库函数，应当在程序开始处用编译预处理命令将包含有库函数定义的相关头文件包含进来。

如要使用输入输出类函数就要用

```
#include<stdio.h>
```

使用数学类函数就要用

```
#include<math.h>
```

各类库函数请参阅附录。

1.3 设计 C 语言程序的基本过程

1.3.1 C 语言程序编程环境

高级语言编写的程序文件称为源代码文件，C 语言编写的源程序文件的扩展名为.C，可以采用各种编辑器编辑 C 语言源程序，如记事本、UltraEdit 等，也可以使用 C 语言编译器的 IDE 环境中自带的编辑器。计算机不能直接识别高级语言程序，要经过编译器编译成二进制的目标代码，生成扩展名为.obj 的文件，然后与相关的库函数连接装配成可执行的代码，生成.exe 文件后才能在计算机上运行。C 语言程序设计的基本过程如图 1.3 所示。

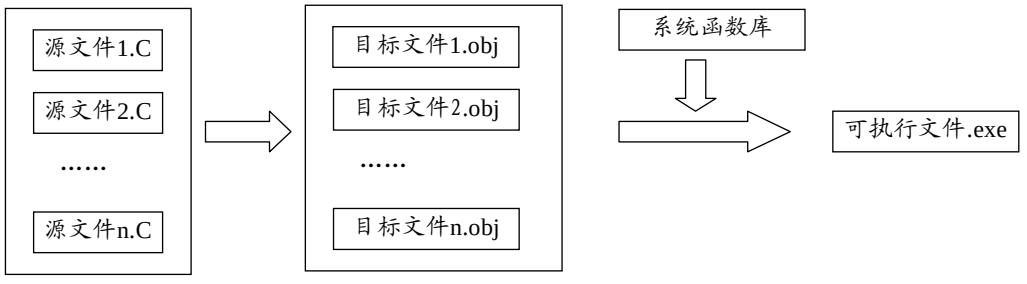


图 1.3 C 程序设计基本过程

现在用 VC++6.0 集成开发工具设计一个 C 语言源程序，然后对其编译、连接，最后运行，查看程序的运行效果。

【例 1.2】写一个最简单的 C 语言程序，在屏幕上输出一个字符串“Hello World!”。

第 1 步：建立源文件，如图 1.4 所示。

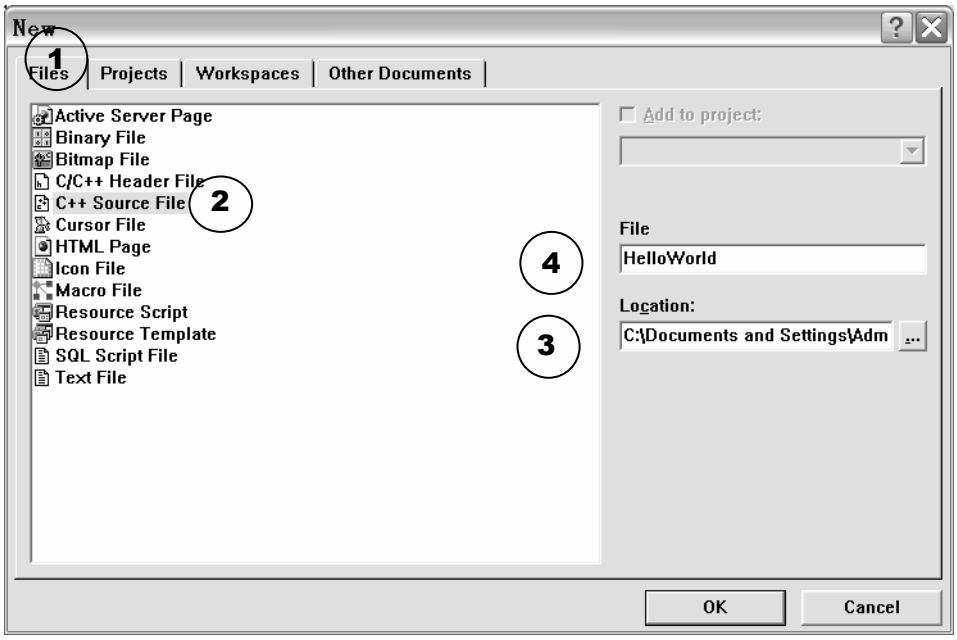


图 1.4　新建源文件窗口

① 选 Files(新建文件)标签。

② 选文件类型为“C++ Source File”。

③ 调整文件存取路径 Location。

④ 填写文件名(默认文件扩展名 CPP，可以加.c)。

第 2 步：编辑源文件，如图 1.5 所示。

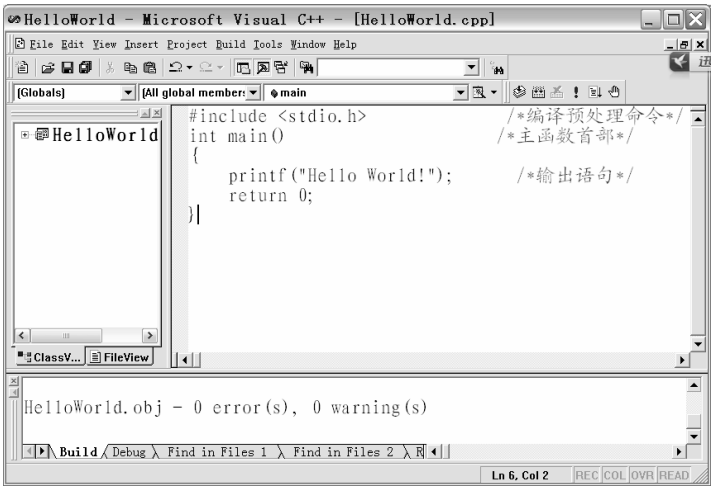


图 1.5　源文件编辑窗口

1.3.2 编译、连接和运行

C 语言程序是否正确的唯一标准就是程序的运行结果与预期结果是否完全相同，在运行程序之前必须先对程序进行编译、连接。

1. 编译、连接和运行 C 程序的步骤

① 编译程序，它的作用是检查程序中是否有语法错误，编译无误可生成目标文件(.obj 文件)。

② 连接目标文件，因为编译后得到的文件是孤立的，仍然不能运行，需要与库文件或程序中的其他文件进行连接，如程序中用到的 printf()函数，它的定义放在库文件中。连接无误后便可生成可执行文件(.exe 文件)。

③ 最后运行可执行文件，得到运行结果。

以上三步通常要借助于编译器，常见的编译器有 TC、Microsoft Visual C++ 6.0、gcc 等。本书全部采用 Microsoft Visual C++ 6.0 作为编译器，具体使用方法详见实验指导。

2. C 程序的错误

在以上的三步中，每一步都可能出现错误，出现错误都需要重新返回编辑环境，修改程序直到程序正确为止。程序中出现的错误可以分为 2 类：语法错误和语义错误。

① 语法错误：在编译和连接阶段出现的错误称为语法错误，这也是初学者经常会遇到的情况，修改语法错误主要借助于编译器，它会提示你错误所在位置及错误的性质和原因。

可以采用倒推的方法总结典型错误，从一个正确的程序出发，将其加以改动，然后编译，查看结果，如果出现错误，查看对应的错误提示。

② 语义错误：程序可以正常运行，但结果与预期结果不同，这种错误称为语义错误或逻辑错误。语义错误的修改主要借助于调试。

3. 调试程序

调试的主要方法有单步跟踪和设置断点两种，这两种方法都要借助于观察变量。调试的原理就是通过对程序中语句的逐条执行或者多条语句一次性执行，对程序错误进行定位。

① 单步跟踪(Trace Step by Step)：即一步一步跟踪程序的执行过程。

② 设置断点(Break Point Setting)：可以在程序的任何一个语句上做断点标记，程序会直接运行到断点所在处并停下来。

③ 观察变量(Variable Watching)：当程序运行到断点的地方停下来后，就可以观察各个相关变量的值，判断此时变量值是否正确，从而判断出程序是否正确，如果发生错误，说明在该处或之前的语句已经出现错误，以此实现对程序错误的定位。

1.4 如何学习 C 语言

学习 C 语言途径很多，但是如果掌握了学习 C 语言的一般规律，会得到事半功倍的效果，不至于越学越累，找不到学习 C 语言的乐趣。下面是给读者推荐的建议。

1. 体会 C 语言严谨性，养成正确逻辑思维习惯

严谨性是 C 语言的重要特征之一。读程序、编写程序，都要一句一句来，注意对程序进行分析，加强逻辑思维锻炼，并以流程图的方式描述出分析的结果，即解决问题的流程。流

程图的绘制方法详见第 3 章。

2. 注意程序的结构——结构化编程思想

应该明确程序目的，按照输入、数据处理和输出三个部分分析问题。其中数据处理是关键部分，只使用顺序结构、分支结构和循环结构这三种基本结构。

3. 用积木搭建的宫殿——模块化编程思想

用积木搭建宫殿，大多数人都非常清楚，其实用 C 语言进行稍大系统设计时，就是通过把大问题分解成小问题(即模块)来实现的。如果读者浏览过第 3 章，就会发现 C 语言的程序其实就是一个一个的模块搭建起来的系统。

4. 掌握能工巧匠的基本技能——调试程序

如果一个程序是建好的宫殿，那么编程者就是建筑宫殿的工匠。要想做一个能工巧匠，除了养成良好的编程习惯外，还要积累丰富的调试程序的技巧和经验。单步执行、设置断点和观察中间变量就是很好的方法,但是如何分析错误和灵活应用这些方法还需要不断的积累。

5. 遵守良好的编程风格、养成良好的编程习惯

如书写语句时注意阶梯缩进，给出详细的注释和必要的空行等，都是好程序源代码具有的风格。初学者如果能从一开始就养成这些好习惯，就能够少犯错误，少走弯路。具体的编程风格请参见附录。

习　题　1

1. 以下不能定义为用户标识符的是_____。

   A. scanf　　B. Void　　C. _3com_　　D. int

2. 计算机能直接执行的程序是_____。

   A. 源程序　　B. 目标程序　　C. 汇编程序　　D. 可执行程序

3. C语言源程序名的后缀是_____。

   A. .exe　　B. .c　　C. .obj　　D. .cp

4. 对于一个正常运行的C程序，以下叙述中正确的是_____。

   A. 程序的执行总是从main函数开始，在main函数结束

   B. 程序的执行总是从程序的第一个函数开始，在main函数结束

   C. 程序的执行总是从main函数开始，在程序的最后一个函数中结束

   D. 程序的执行总是从程序中的第一个函数开始，在程序的最后一个函数中结束

5. 以下叙述中正确的是____。

   A. C程序的注释部分可以出现在程序中任意合适的地方

   B. 大括号“{”和“}”只能作为函数体的定界符

   C. 构成C程序的基本单位是函数，所有函数名都可以由用户命名

   D. 分号是C语言之间的分隔符，不是语句的一部分

6. 在一个C程序中____。

   A. main函数必须出现在所有函数之前　　B. main函数可以在任何地方出现

   C. main函数必须出现在所有函数之后　　D. main函数必须出现在固定位置

7. 以下4组用户定义的标识符中，全部合法的一组是____。

A. _main
enclude
sin

B. If
−max
turbo

C. txt
REAL
3COM

D. int
k_2
_001

8. 一个算法应该具有“确定性”等5个特性，下面对另外4个特性的描述中错误的是____。

A. 有零个或多个输入

B. 有零个或多个输出

C. 有穷性

D. 可行性

9. 以下叙述中正确的是____。

A. C语言的源程序不必通过编译就可以直接运行

B. C语言中的每条可执行语句最终都将被转换成二进制的机器指令

C. C语言源程序经编译形成的二进制代码可以直接运行

D. C语言中的函数不可以单独进行编译

第 2 章　数据类型和表达式

本章主要介绍 C 语言中的数据类型，输入、输出方法和常用运算符。

程序中处理的所有数据分为常量和变量，变量需要声明，声明的过程就是确定它的数据类型和名字；接下来就是对变量进行处理，主要借助于各种运算符。运算符与变量或常量进行有意义的　合后就形成了表达式。在运算的过程中，如果要引用变量的值，需对变量进行初始化，常用的初始化方法有使用赋值运算符和标准输入，　后将运算的结果输出到屏幕等标准输出设备上。C 语言中主要借助于 scanf()和 printf()库函数实现标准输入和输出。

第一次阅读本章时，以下内容可以采用迅速阅读的方式，即不必求理解，或者直接越过，待本章全部阅读完后再重新看过，　为检验你学习效果的标准。

1. 掌握不同数据类型之间的区别，即 int，float，double 与 char 之间的区别。

2. 掌握输入输出函数的语法结构及使用方法，即注意函数参数的意义及整个函数所代表的含义。

3. 掌握常用运算符的运算规律。常用运算符主要有：算术运算符、关系运算符、赋值运算符和逻辑运算符。

知 识 结 构

1. **C 语言基本数据类型**
   - ① 整型 int
   - ② 实型 float、double
   - ③ 字符型 char
2. **常量**
   - ① 整型常量
   - ② 实型常量
   - ③ 字符型常量
   - ④ 符号常量
3. **变量**

   变量的定义
4. **数据的输入与输出**
   - ① 数据的输出：printf()函数、putchar()函数
   - ② 数据的输入：scanf()函数、getchar()函数
5. **运算符**

   算术运算符、关系运算符、逻辑运算符、赋值运算符、条件运算符、逗号运算符

6. 数据类型转换

自动转换与强制转换

2.1 C 语言数据类型

C 语言中数据类型可分为：基本数据类型、构造数据类型、指针类型、空类型四大类。

1. 基本数据类型

基本数据类型最主要的特点是其值不可以再分解为其他类型。基本数据类型有以下 4 种：

① char：字符型。

② int：整型。

③ float：单精度浮点型。

④ double：双精度浮点型。

整型数据是不带小数点的数据，它在计算机中以补码形式存储数值。字符型数据在计算机中存储的是它对应的 ASCII 码的二进制形式(见附录Ⅰ)。浮点型数采用与整型数据完全不同的存储方式，这不属于 C 语言的范畴，在计算机组成原理等相关课程中会进行详细介绍。

此外在以上某些数据类型前可以加限定符，如在 int 前可以加 short 和 long，用来限定数据的存储长度，进而改变它可表示的数值范围。限定符 unsigned 代表无符号数，可以对 char 型数据和 int 型数据进行限制，无符号数即该数没有符号位，原来的首位可以用来表示数值，从而扩大数的表示范围。

C 语言中的基本数据类型的类型说明符、所占字节数及取值范围如表 2.1 所示。

表 2.1 C 语言基本数据类型

类 型	类型说明符	字节	数值范围
基本整型	int	4	$-2147483648\sim2147483647$ ($-2^{-31}\sim2^{31}-1$)
短整型	short int	2	$-32768\sim32767$ ($-2^{-15}\sim2^{15}-1$)
长整型	long int	4	$-2147483648\sim2147483647$ ($-2^{-31}\sim2^{31}-1$)
无符号型	unsigned	4	$0\sim4294967295$ ($0\sim2^{32}-1$)
无符号短整型	unsigned short	2	$0\sim65535$ ($0\sim2^{16}-1$)
无符号长整型	unsigned long	4	$0\sim4294967295$ ($0\sim2^{32}-1$)
字符型	char	1	$0\sim255$
单精度实型	float	4	$\pm(10^{-38}\sim10^{38})$
双精度实型	double	8	$\pm(10^{-308}\sim10^{308})$

其中类型说明符是在定义变量类型和声明函数返回值类型等时采用的符号，属于 C 语言的关键字。类型所占字节数及取值范围与计算机硬件系统和所采用的编译系统有关，可作为编程时选择数据类型的依据。

C 语言中存放在内存中的数据除了类型不同之外，根据它们的值在程序运行过程中是否可以发生变化，又可将其分为变量和常量。

2. 构造数据类型

构造数据类型是根据已定义的一个或多个数据类型用构造的方法来定义的。也就是说，

一个构造类型的值可以分解成若干个“成员”或“元素”。每个“成员”都是一个基本数据类型或又是一个构造类型。在 C 语言中，构造类型有以下几种。

① 数组类型：

例如定义语句“int a[10];”表示 a 是一个整型数组，长度为 10，从 a[0]到 a[9]共 10 个数组元素。在内存中开辟了连续 10 个 int 类型数据的空间，如果 int 类型数据占 4 个字节，那就是开辟了连续的 40 个字节的空间来存储 10 个整型数组元素。数组可分为一维数组、二维数组和多维数组。

② 结构体类型：

它和数组类型的相同点是它也是将多个数据定义成一个整体，不同点是结构体类型的成员可以是不同数据类型的数据。如下例中定义了一种新的结构体类型即 struct person 类型，它的成员变量 name 数组、age、sex 等分属不同的数据类型。详见第 7 章。

```
struct person
{
   char name[10];
   int age;
   char sex;
   char telephone[10];
};
```

③ 共用体类型：详见第 7 章。

3. 指针类型

指针是一种特殊的、具有重要作用的数据类型，其特殊性体现在指针的值代表某个变量在内存中的地址。虽然指针变量的取值类似于整型量，但它们是两个类型完全不同的量，因此不能混为一谈。

4. 空类型

空类型即 void 类型，经常用于表示函数返回值的类型。

本章主要介绍基本数据类型，后续相关章节会对其余类型做详细介绍。

2.2　变　　量

变量就是程序在运行过程中其值可以变化的量。变量代表内存中具有特定属性的一个存储单元，变量的使用及取值范围等由它的类型决定，因此所有的变量在使用之前必须定义。

变量定义方式如下：

```
数据类型说明符 变量名 1, 变量名 2, … , 变量名 n;
```

如：

```
int i, j, k;
float radius, height;
double volume;
char ch;
```

① 数据类型说明符决定了变量所存储的数据种类，可根据实际情况进行选择，可以是 int、float、char 等基本数据类型，也可以是上面提到的 struct person 等构造数据类型。

② 编写程序时需要对变量、函数、数组和其他实体使用自定义标识符进行命名。编译系统会根据定义变量时说明的数据类型给变量分配相应大小的存储空间，变量名实际上是该存储空间的别名。

③ 变量定义语句通常书写在函数体的开头位置，定义同类型的多个变量时，变量之间可以用逗号分隔，定义语句末尾以分号结束。

【练习 2.1】已知半径，求一个圆的面积，应如何书写相关变量的定义语句？

【提示】需要保存圆的半径、面积的数值，而半径和面积都是可以变化的，因此需要定义两个变量，二者均可为小数，特别是面积，因此可将其定义为 float 类型(double 类型也可)。

【练习 2.2】分析以下程序段是否有错误，可编译后查看编译结果，若有错误，查看对应的错误类型和原因。

```
#include<stdio.h>
int  main()
{
   printf("输出 n 的值：");
   int  n;
   n=5;
   printf("n=%d", n);
   return  0;
}
```

【　结】

① 变量使用之前必须先定义。

② 所有变量的定义语句要放在函数体的开头部分。

③ 一个字符型变量只能存放一个字符，因为它只对应一个字节的空间，存放的是该字符的 ASCII 码。

单精度浮点型(float)和双精度浮点型(double)变量除了取值范围的区别外，还有数据精度的区别。在 VC++ 6.0 中，单精度型变量占 4 个字节(32 位)内存空间，只能提供 7 位有效数字，其数值范围为±(10^{-38}～10^{38})。双精度型占 8 个字节(64 位)内存空间，可提供 16 位有效数字，其数值范围为±(10^{-308}～10^{308})。

【练习 2.3】运行以下程序段，验证程序的输出结果是否与 a，b 的初值相同。

```
#include<stdio.h>
int  main()
{
    float  a;
    double  b;
    a=1234567.89;
    b=1234567890123.45678;
    printf("%f, %f\n", a, b);
    return  0;
}
```

2.3 常　　量

常量是在程序运行期间，其值不能发生变化的数据量。常量的类型不需要定义，计算机在编译时可以根据其书写方式自动确定其数据类型，并据此为其分配相应字节大小的空间进行存储。下面主要介绍各种不同类型常量的书写规范及代表含义。

2.3.1 整型常量

整型常量是不带小数点的数，如 18，23，07，0x3a 等。整数常量可以根据情况在其表示代表数值的数码前面加前缀，也可加后缀，其中前缀一般代表进制，后缀代表类型。

1. 前缀

C 语言中共有 3 种不同进制的整型常量：十进制整数、八进制整数、十六进制整数。不同进制的数使用不同的前缀来区分。

① 十进制整数：十进制整数没有前缀。其数码为阿拉伯数字 0～9，不能用 0 作数字开头。

以下各数是合法的十进制整型数：

237，−568，65535，1627

② 八进制整数：以数字 0 开头，即以 0 作为八进制数的前缀。其数码为阿拉伯数字 0～7。

以下各数是合法的八进制整型数：

015(十进制为 13)，0101(十进制为 65)，0177777(十进制为 65535)

以下各数不是合法的八进制整型数：

256(无前缀 0)，03A2(包含非八进制数码 A)，0586(八进制数不能包含 8)

③ 十六进制整数：以数字 0 和字母 x 或字母 X 开头，即以 0x 或 0X 为十六进制整数的前缀。其数码为阿拉伯数字 0～9，英文字母 A～F 或 a～f。

以下各数是合法的十六进制整型数：

0X2A(十进制为 42)，0XA0(十进制为 160)，0XFFFF(十进制为 65535)

以下各数不是合法的十六进制整型数：

5b(无前缀 0x)，0X5H(含有非十六进制数码 H)

【练习 2.4】① 20、020、0x20 是否代表同一数值，如果不是，它们的数值分别为多少？

② 8、010、0x8 是否代表同一数值，如果不是，它们的数值分别为多少？

③ C 语言中是否有二进制常量？

2. 后缀

整型常数的后缀通常用来表示整数的类型。

① 字母 L 或 l，代表 long 型常量。

整型常量作为整型数据，首先必须遵循的规则就是所表示的数不能超出整型的数值范围(见表 2.1)。如果整型常量要表示的数值超过了上述范围，就必须用后缀“L”或“l”来表示长整型数。例如：

十进制长整数 158L(为十进制数 158)

八进制长整数 077L(为十进制数 63)　　0200000L(为十进制数 65536)

十六进制长整数 0X15L(为十进制数 21)　　0X10000L(为十进制数 65536)

② 字母 u 或 U 代表 unsigned(无符号)型常量。

例如：358u，236u 均为无符号数。

也可同时使用前缀和后缀以表示各种类型的数。如 0XA5Lu 表示十六进制无符号长整数 A5，对应的十进制数为 165。

【 结】对于整型常量(即在程序中书写的不带小数点的常量),在代表数值的数码之前或之后可以加上一定符号:

① 前缀 0 代表 8 进制。

② 前缀 0x 或 0X 代表 16 进制。

③ 后缀 L 或 l 代表 long。

④ 后缀 u 或 U 代表 unsigned。

2.3.2 实型常量

实型常量即带小数点的数，也称为浮点数。对于实型常量，系统自动默认其为 double 类型。实型常量有两种表示形式。

1. 十进制数形式

由数码 0～9 和小数点组成。例如：0.0，.26，1.28，5.0，300. ，-267.8230 等均为合法的实型常量。

2. 指数形式

由十进制数加阶码标志“e”或“E”以及阶码(只能为整数，可以带符号)组成。其一般形式为 aEn，其中 a 为十进制数，n 为十进制整数，其值为 $a\times10^n$。如：

2.1E5(表示 2.1×10^5)，2.9E-2(表示 2.9×10^{-2})，-2.8E+5(表示-2.8×10^5)

而以下均不是合法的实型常量：

E7(阶码标志 E 之前无数字)，53E3.0(阶码不能为小数)，2.7E(无阶码)

【 结】

① 带小数点的合法数即为实数。

② 小数点之前和之后均可没有数码，也可同时省略，但小数点不可省略。

③ 指数形式的实型常量的特征是包含 E 或 e。

④ 阶码标志的两侧都不能为空。

⑤ 阶码标志的右侧必须为整数。

⑥ 实型常量的数据类型均为 double 类型。

2.3.3 字符型常量

用单引号括起来的一个字符。例如 'a'、'0'、'='、'+'、'\n'都是合法字符常量。在 C 语言中，字符常量有以下特点。

① 字符常量必须用单引号括起来，不能用双引号或其他符号。

② 除转义字符外，单引号里面只能是单个字符。

③ 转义字符经常用来表示在程序中无法用键盘输入的控制代码或者在程序中另有含义的特殊字符。

④ 每个字符都与它的 ASCII 码一一对应，二者是等价的，如 '1'的 ASCII 码是十进制的 49，因此字符 '1'在参与运算时，就表示 49 在参与运算。具体请见附录Ⅰ。

⑤ 要注意的是，在程序中 'A'表示一个字符常量，而 A 表示标识符。

'a'，'1'等都可以通过键盘输入到程序中，但回车符、退格符等字符无法通过键盘输入到程序中，有的也无法输出显示在屏幕上，或者如分号等字符在 C 语言语法中具有特殊的含义，对于以上各类字符，都可以用其转义字符的形式表示，常见的转义字符如表 2.2 所示。

转义字符以反斜线“\”开头，后跟一个或几个字符。转义字符具有特定的含义，不同于字符原有的意义，故称“转义”字符。例如，在前面各例题 printf()函数的格式串中用到的“\n”就是一个转义字符，其意义是“回车换行”。

表 2.2　常见转义字符及其含义

转义字符	转义字符的含义	ASCII 码
\n	回车换行，将光标移到下一行的开始位置	10
\t	光标移到下一个水平制表位置	9
\b	退格	8
\r	回车，将光标移到当前行的开始位置	13
\"	双引号	34
\'	单引号	39
\\	反斜线符 \	92
\ddd	1～3 位八进制数所代表的字符	如\141 代表'a'
\xhh	1～2 位十六进制数所代表的字符	如\x41 代表'A'

广义地讲，C 语言字符集中的任何一个字符均可用转义字符来表示，表 2.2 中的“\ddd”和“\xhh”可以实现这点。ddd 和 hh 分别为八进制和十六进制的 ASCII 代码。如“\101”表示字母'A'，“\102”表示字母'B'，“\134”表示反斜线，“\XA”表示回车等。

【练习 2.5】① 判断下述字符常量"a"、'ch'、"\n"、"\0x23"、'\025'、'\223'、'x41'的合法性。

② 如下程序段的输出结果为_____。

```
#include <stdio.h>
int  main()
{
      int  i;
      char  c='2';
      i=c+2;
      printf("%d %d", c, i);
      return  0;
}
```

2.3.4　符号常量

在 C 语言中，可以用一个标识符来表示一个常量，称之为符号常量。符号常量在使用之前必须先定义，其一般形式为：

```
#define  标识符  常量
```

其中#define 也是一条预处理命令(预处理命令都“#”开头)，称为宏定义命令(详见第 7 章)，其功能是把该标识符定义为其后的常量值。一经定义，以后在程序中所有出现该标识符的地方均代之以该常量值。

习惯上符号常量的标识符均用大写字母，变量标识符用小写字母，以示区别。

【例 2.1】分析以下程序段。

```
#include <stdio.h>
#define  PI  3.14159
int  main()
{
      float  s, r=5;
      s=PI*r*r;
      printf("s=%f\n", s);
      return  0;
}
```

本程序在主函数之前由宏定义命令定义 PI 为 3.14159，程序在编译之初，将程序段中出现 PI 的地方均用 3.14159 来代替，程序在编译之前被替换成如下程序段。

```
#include<stdio.h>
int  main()
{
      float  s, r=5;
      s=3.14159*r*r;
      printf("s=%f\n", s);
      return  0;
}
```

注意符号常量和变量之间的区别，前者所代表的值在整个程序运行期间都不能改变。如不能再用赋值语句对符号常量进行赋值。

【思考】使用符号常量有什么好处？

2.4 数据的输入、输出

C 语言中主要借助于库函数来实现数据输入输出，经常使用的如格式化输入函数 scanf() 和格式化输出函数 printf()。

2.4.1 printf()函数

printf()函数用来以特定的格式在屏幕上输出给定的内容。具体格式如下：

printf(格式控制字符串，输出参数 1, …，输出参数 n);

其中，格式控制字符串是以英文双引号括起的一段字符串，用于指定输出格式。

格式控制字符串由两类字符组成：格式控制说明符和非格式控制说明符。

格式控制说明符特点是以%开头，常用的格式控制说明符有%d、%f、%lf、%e、%c、%s 等，用来说明输出数据的类型、进制、长度、输出位数等，格式控制说明符控制的是它所对应的输出参数的值，因此输出参数要与格式控制说明符一一对应，包括类型、位置与数量都要一致。输出参数可以是常量、变量或者任意合法的表达式，输出的格式由格式控制说明符决定。非格式控制说明符按照原样输出。

【例 2.2】print()函数使用示例。

```
#include<stdio.h>
int  main()
{
   int  c;
   c=1;
   printf("c=%d", c);
   return  0;
}
```

本例调用 printf()输出函数，“c=%d”为格式控制字符串，其中“c”、“=”均为普通字符，按原样输出，“%d”为格式控制说明符，以十进制整型格式输出后面输出参数变量 c 所对应的值，即 1。因此，输出结果为：

```
c=1
```

函数 printf()格式控制符如表 2.3 所示。

表 2.3　**printf()函数的格式控制符**

格式控制符	说　　明
%d	以十进制形式输出整数(正数不输出符号)
%o	以八进制形式输出整数(不输出前导符 0)
%x	以十六进制形式输出整数(不输出前导符 0x 或 0X)
%u	以无符号十进制形式输出整数
%c	以字符形式输出一个字符
%s	输出字符串
%f 或%lf	以小数形式输出浮点数，保留小数点后 6 位
%e 或%le	以指数形式输出浮点数
%m.nf	m，n 为正整数，输出浮点数总宽度为 m，包括小数点，保留 n 位小数

1. 整型数据的输出

%d 为十进制整型数据的格式控制说明符，十六进制和八进制的整型数据格式控制说明符如表 2.3 所示，此外附加格式说明符 l 用于长整型整数，可加在格式字符 d、o、x、u 前面。

例如语句“printf("a=%d, b=%d", 2, 3);”中，双引号"和"里的 a=、b=是用于说明的信息，而%d 则是格式转换说明，用于指定输出数据的格式为整型，第 1 个%d 对应第 1 个数字，第 2 个%d 对应第 2 个数字，因此输出在计算机屏幕上的结果为：a=2, b=3。

可以用 m(一个正整数)指定输出宽度，如果宽度大于数据实际位数，则左端补空格，如果小于实际位数则按照实际位数输出。

2. 实型数据的输出

输出实型数据时采用的格式说明符有%f、%lf 和%e，其中%e 表示以指数形式输出，%f 和%lf 表示以小数形式输出，%lf 主要用于输出双精度浮点型数据(double)。以小数形式输出时，默认为小数点后 6 位，如果要加以限制可采用加宽度限定词的扩展形式，即%m.nf 的形式，表示保留 n 位小数，且输出宽度为 m 位(包括符号位和小数点)，如果数据的实际位数小于 m，左端补空格，若大于 m，则按实际位数输出。

【例 2.3】分析下面程序段，判断输出结果。

```
double  pi=3.141592;
```

```
printf("%5.3f%5.2f", pi, pi);
```

输出结果为：

```
3.142  3.14
```

3. 字符型数据的输出

字符型数据输出采用的格式控制说明符为%c，如下所示。

```
printf("c=%c", c);
```

输出字符型数据除了可以用 printf()函数外，还可用 putchar()函数。

putchar()函数可以输出一个字符，它只有一个参数，可以是字符型常量，也可以是字符型变量。

【练习 2.6】分析以下程序段输出结果。两次调用 putchar()函数的输出结果相同吗？

```
char  c='d';
putchar(c);
putchar('c');
```

【练习 2.7】已知英文字母 'a'的 ASCII 码为 97(十进制)，则以下程序段的输出结果为：_____。

```
char  c='0', b='j', a='a';
printf("%d, %c, %x, %X, %o", c, c, b, b, a);
```

【提示】c 中存的是数字字符'0'。八进制和十六进制的常数都有前缀，但在以八进制或十六进制格式进行数据的输出时，不会输出前缀。

2.4.2 scanf()函数

1. 变量的初始化

【例 2.4】分析以下程序段的输出结果。

```
#include<stdio.h>
int  main()
{
   int  c;
   printf("c=%d", c);
   return  0;
}
```

程序运行后输出结果为随机值，说明变量 c 的值是随机值，这是为什么呢？因为在定义变量时，只是为变量开辟了相应的内存存储空间，并未确定变量的初值，由系统随机分配，这样的随机值显然无法使用。因此所有的变量在引用之前除了要定义外，还必须赋初值。

赋初值的方法有两种，第 1 种是直接用赋值运算符“=”赋值，此时“=”的左侧必须是变量，将右侧表达式的值赋值给左侧的变量。第 2 种方法是在程序运行时，由用户通过键盘给变量赋值，这种方法可以通过调用格式化输入函数 scanf()实现。

2. scanf()函数说明

格式化输入函数 scanf()的作用是给变量赋初值。其格式与 printf()函数类似：

```
scanf(格式控制字符串, 输入参数 1, … , 输入参数 n);
```

它与 printf()函数不同之处在于：

① 格式控制字符串中除格式控制说明符之外的其他字符要原样输入。

② 作为输入参数的变量名前要加取地址运算符&。

scanf()函数的作用是以格式控制说明符代表的格式将键盘输入的内容存入到输入参数所代表的存储单元中，输入参数通常以&变量名的形式给出。

【例 2.5】分析以下程序段的输出结果。

```
#include<stdio.h>
int  main()
{
    int  c;
    scanf("%d", &c);
    printf("c=%d", c);
    return  0;
}
```

运行实例：

```
2
c=2
```

【思考】① 如果把上面程序段中的变量 c 定义为 char 类型，输入不变，分析输出结果。

② 将变量 c 定义为 char 类型，输入不变，将 scanf()修改为如下格式，分析输出结果。

```
scanf("%c", &c);
```

字符型数据在输入时除了可以使用 scanf()函数外，还可以使用 getchar()函数。格式为：

```
getchar();          // 无参数，函数的返回值为从键盘输入的一个字符
```

例如语句：

```
ch=getchar();
```

用就是将通过键盘输入的一个字符赋值给变量 ch。

使用键盘输入多个字符型数据时，输入的字符之间不能有间隔。如果使用了间隔符(如空格或回车等)，由于它们本身也是字符，该间隔符就被　为输入字符赋值给相应变量。

【例 2.6】对于如下程序，怎样输入才能得到输出结果 digit=2，letter=Ae。

```
#include<stdio.h>
int  main()
{
    char  digit, letter;
    scanf("%c", &digit);
    letter=getchar();
    printf("digit=%c, letter=%c", digit, letter);
    putchar('e');
    return  0;
}
```

运行程序，输入 2 并按回车键后，程序已结束，输出结果为：

```
digit=2，letter=
e
```

这是因为输入回车键也相当于输入了一个字符，字符 2 赋值给了变量 digit，回车符赋值给了变量 letter。因此只有输入 2A 后回车，才能得到“digit=2，letter=Ae”的输出结果。

scanf()函数的格式字符如表 2.4 所示。

表 2.4　scanf()函数的格式字符

格式控制符	说　　明
%d	用于输入十进制整数
%o	用于输入八进制整数
%x	用于输入十六进制整数
%c	用于输入单个字符
%s	用于输入字符串(以空格或回车结束的一串字符)
%f 或%e	用于输入单精度浮点数(float 类型)
%lf 或%le	用于输入双精度浮点数(double 类型)

【　结】scanf()函数的格式控制说明符大部分与 printf()函数相同，不同的地方如下:

① 输入 double 类型数据时，格式控制说明符必须为%lf，不可以使用%f。

② 输入实型数据时，不可以添加宽度限定词，即%m.nf 的形式只能在输出时使用。

③ scanf()的输入参数必须代表地址，方式为&变量名，而 printf()函数的参数可以是变量、常量以及任意合法的表达式。

④ scanf()函数的格式控制字符串中的普通字符必须按原样输入。

如执行

```
scanf("%d, %d", &a, &b);
```

需要输入“3, 4 回车”，第一个数据的后面必须紧跟逗号，因为"%d, %d"中有逗号。

又例如执行

```
scanf("a=%d, b=%d", &a, &b);
```

需要输入“a=3, b=4 回车”，为了减少不必要的输入错误，在 scanf()函数的格式控制字符串中尽量不出现普通字符。

例如执行

```
scanf("%d %d", &a, &b);
```

输入多个数据时，以空格或者回车隔开即可，可以输入“3 回车，4 回车”，或者“3　4 回车”。

【练习2.8】有如下程序:

```
#include<stdio.h>
int  main()
{
   int  m, n, p;
   scanf("m=%d,n=%d,p=%d", &m, &n, &p);
   printf("%d%d%d\n", m, n, p);
   return  0;
}
```

若想从键盘上输入数据，使变量 m 中的值为 123，n 中的值为 456，p 中的值为 789，则正确的输入是_____。

A. m=123n=456p=789　　B. m=123　n=456　p=789

C. m=123,n=456,p=789　　D. 123　456　789

2.5　运算符与表达式

C 语言提供了丰富的运算符，用运算符和操作数做有意义的组合就构成了表达式。最简单的表达式是常量、变量和函数。C 语言中不同的运算符构成不同的表达式，因此掌握表达式主要是掌握运算符，包括运算符的优先级和结合性，如表 2.5 所示。

表 2.5　部分运算符的优先级和结合性

运算符类别	运 算 符	结 合 性	优先级
逻辑运算符	!	从右向左(右结合)	高
算术运算符	++、--、+、-（单目）	从左向右(左结合)	↑
	*、/、%　（双目）		
	+、-（双目）		
关系运算符	<、<=、>、>=		
	==、!=		
逻辑运算符	&&		
	||		
条件运算符	?:（三目）	从右向左(右结合)	
赋值运算符	=、+=、-=、*=、/=、%=		
逗号运算符	,	从左向右(左结合)	低

使用频率最高的是算术运算符、关系运算符、逻辑运算符和赋值运算符。

单目运算符，又称为一元运算符，表示只有一个操作数，双目运算符和三目运算符以此类推。

优先级表示运算的先后顺序。优先级高的运算符应先进行运算，反之后进行运算。

同一优先级的运算符，运算次序由结合性决定。结合性表明运算时的结合方向，分为自左向右和自右向左，一般情况下单目运算符和三目运算符为自右向左，双目运算符为自左向右。

在运算时为更好明确优先级和结合性，可以通过加圆括号的方式改变计算顺序。

2.5.1　算术运算符

根据所需运算量的个数将算术运算符分为两类：双目运算符和单目运算符，其中前者主要有加+(加)、-(减)、*(乘)、/(除)和%(求余)5 个运算符，均为左结合性运算符；后者主要有+(正)、-(负)、++(自增)、--(自减)4 个运算符，均为右结合性运算符。算术运算是最常用的运算，C 语言的算术运算与普通数学中的运算有一定的差异，应注意区别。

① 加法运算符+、减法运算符-：如 a+b，b-8，+、-运算符也可作单目运算符表示正负性，如-x、+7 等。

② 乘法运算符*：在数学中 a*b 可以简写为 ab，但 C 语言中不可以简写，只能写成 a*b，因为简写成 ab 后就成为一个标识符了。

③ 除法运算符/：参与运算量均为整型时，结果也为整数，即舍掉商的小数部分，不采用四舍五入商。如 1/2 的值为 0，而非 0.5。如果运算对象中有一个是实型，则结果为实型。

【例 2.7】分析以下程序段的输出结果。

```
#include<stdio.h>
int  main()
{
   int  i=3;
   printf("%d, %d, %f\n", i/2, 1/i, 1.0/i);
   return  0;
}
```

以上程序的运行结果为：

1, 0, 0.333333

④ 求余运算符(模运算符)%：要求两个操作数均为整型。求余运算的结果等于两数相除后的余数。一般情况下，余数的符号与被除数符号相同。例如 5%3 结果为 2，-8%5 结果为 -3。

⑤ 自增运算符++、自减运算符--：针对变量的一种单目运算符，具有右结合性。作用是使变量的值增 1 或减 1。

++运算符和--运算符既可以作为前缀运算符(如++i，--j)，又可以作为后缀运算符(如 i++，j--)，不管是作为前缀还是后缀，对变量本身的值而言都是增 1 或减 1，所不同的是表达式的值。

以自增为例，++i 和 i++作用都相当于 i=i+1，不同处在于++i 是先执行 i=i+1 后，再取 i 的值作为表达式的值；而 i++是先取 i 的值作为表达式的值，再执行 i=i+1。

例如，i1，i2 的初值都为 3，则执行下面的赋值语句：

```
j1=++i1;      // 结果：j1=4，i1=4
j2=i2++;      // 结果：j2=3，i2=4
```

例如执行下述程序段：

```
i=3;
printf("%d", ++i);
```

输出结果为 4。若改为

```
printf("%d", i++);
```

则输出结果为 3。

所以++i 意味着“立即自增 1”，而 i++意味着“先用 i 原来的值，稍后再自增 i”。

自增、自减运算符本身的运算规则比较简单，但在分析程序时首先应区分的就是：要计算或输出的是变量的值，还是表达式的值。

自增、自减运算符经常用于循环语句中，详见第 3 章。

【注意】自增、自减运算符的运算对象只能为变量，不能是常量或表达式。思考一下原因，这个问题可在学完赋值运算符后再回答。

【练习 2.9】① 执行程序段“int i=3;   printf("%d", i--);”输出结果为_____。

A. 3　　B. 4

② 以下不能正确表达 $\frac{2ab}{cd}$ 的C语言表达式的是_____。

A. 2*a*b/c/d　　B. a*b/c/d*2　　C. a/c/d*b*2　　D. 2*a*b/c*d

2.5.2　关系运算符

C 语言中的关系运算符均为双目运算符，主要有以下 6 种：

<(小于)、>(大于)、<=(小于等于)、>=(大于等于)、==(等于)、!=(不等于)

关系运算符的优先级低于算术运算符，高于赋值运算符。关系运算符都是双目运算符，其结合性均为左结合。

<、>、<=、>=四个运算符优先级相同，高于==、!=运算符，==、!=运算符的优先级也相同。

关系运算符用于判断参与运算的左右两个运算对象是否满足运算符表示的关系，如是，则结果为真，否则为假，因此由关系运算符构成的关系表达式的值只有 1(真)或 0(假)两个。关系表达式在参与其他算术运算时，真即 1，假即 0，而数字中无论是正数还是负数，只要不是 0，都被看成真，只有 0 被看成假。

关系运算符经常用于构成条件表达式，在 if 语句、for 语句、while 语句、do…while 语句中表示条件(详见第 3 章)。同时要注意 C 语言中关系表达式与数学中代数式之间的区别

【例 2.8】分析下列程序段的输出结果，结果为“No”吗？

```
#include<stdio.h>
int  main()
{
   int  x=7;
   if(3<=x<=5)
        printf("Yes");
   else
        printf("No");
   return  0;
}
```

所以以上程序段的输出结果为“Yes”，而不是“No”，原因如下：

在数学中，3<=x<=5 式子的作用是用于判断 x 是否在 3 和 5 之间，如是，结果为真，否则为假。而在 C 语言中，它是一个关系表达式，采用左结合的运算方式，运算顺序为自左向右，首先判断 3<=x 是否为真，然后再判断该结果与 5 之间的关系。无论 x 的值为多少，作为关系表达式的 3<=x 结果非 0 即 1，而 0 和 1 都小于 5，因此在 C 语言中 3<=x<=5 表达式的值永真。在 C 语言中判断 x 是否在 3 和 5 之间要借助于下面介绍的逻辑运算符。

2.5.3　逻辑运算符

C 语言中的逻辑运算符有 3 个，分别为：&&(逻辑与)、||(逻辑或)、!(逻辑非)。用逻辑运算符将关系表达式或逻辑量连接起来构成的表达式称为逻辑表达式。

&&和||为双目运算符，!为单目运算符。逻辑运算对象可以是关系表达式或逻辑量，也可以是任何一个有意义的表达式，运算时都要将其转化为逻辑量，即真或假。转化方法是：无论是正数还是负数，只要不是 0，都被看成真，只有 0 被看成假。它的运算结果也只有 2 种：真和假。结果为真用 1 表示，结果为假用 0 表示。

逻辑运算真值表如表 2.6 所示。

表 2.6　逻辑运算真值表

a	b	!a	a&&b	a||b
真(非 0)	真(非 0)	0	1	1
真(非 0)	假(0)	0	0	1
假(0)	真(非 0)	1	0	1
假(0)	假(0)	1	0	0

如果要判断 x 的范围是否在 3 和 5 之间，则应如下表示：

(x>=3)&&(x<=5)

【注意】① 在求表达式 exp1&&exp2 的值时，如果 exp1 的值为假，则不再对 exp2 进行计算。

② 同样道理，在求解表达式 exp1||exp2 的值时，如果 exp1 的值为真，则不再对 exp2 进行计算。

③ ！运算符的优先级高于所有的算术运算符，“&&”运算符优先级低于算术运算符的优先级，“||”运算符的优先级低于&&运算符。

【练习2.10】① 已有定义“char c=' '; int a=1, b;”(此处c的初值为空格字符)，执行语句“b=!c&&a;”后b的值为_____。

② 执行以下程序段后，w的值为_____，x的值为_____。

```
int w='A', x=10, y=15;
x=((x&&y)||(w='B'));
```

2.5.4　赋值运算符

“=”是赋值运算符，由“=”连接的式子称为赋值表达式。其一般形式为：

变量=表达式

赋值运算符的左侧必须是变量，不能是常量，也不能是表达式。它的作用是将右侧表达式的值赋值给左侧变量。

① 赋值运算符的优先级低于算术运算符、关系运算符、逻辑运算符。

【例 2.9】a，b 的值分别为 0 和 1，求经过以下运算后 w，x，y 的值：

w=a+b;　x=a>b;　y=a++ + --b;

根据以上原则可得，w=1；x=0；y=0。

② 赋值运算符的运算顺序是自右向左，右结合。

【例 2.10】若 c 的值为 0，求表达式 a=b=c>5 的值。

此表达式可理解为 a=(b=(c>5))，即 a 的值为 0。

③ 复合赋值符及表达式。

在赋值符=之前加上其他二目运算符可构成复合赋值符。如：

+=、-=、*=、/=、%=

构成复合赋值表达式的一般形式为：

变量 双目运算符=表达式

它等价于

变量=变量 运算符 表达式

例如：a+=5 等价于 a=a+5。

x*=y+7 等价于 x=x*(y+7)。

r%=p 等价于 r=r%p。

【练习2.11】① 若变量均已正确定义并赋值，以下语句中，合法的C语言赋值语句是_____。

A. x=y= =5;　　B. x=n%2.5;　　C. x+n=i;　　D. x=5=4+1;

② 设变量a和b已正确定义并赋初值。与a−=a+b等价的赋值表达式为_____。

2.5.5　条件运算符

条件运算符“？：”是 C 语言中唯一的一个三目运算符，它的优先级低于逻辑运算符，结合性是右结合，即从右向左。一般形式为：

表达式 1?表达式 2:表达式 3

表达式 1、表达式 2、表达式 3 可以是任意类型的表达式。

条件运算符的运算规则如下：如果表达式 1 的值为真，那么整个条件表达式的值为表达式 2 的值，否则为表达式 3 的值。以条件运算符构成的表达式称为条件表达式。

条件运算符的运算规则可以用第 3 章中的分支结构中的 if()~else 语句来实现。

【练习2.12】运行以下程序后的输出结果是_____。

```
#include<stdio.h>
int  main()
{
   int  x, a=1, b=2, c=3, d=4;
   x=(a<b)? a:b;     x=(x<c)? x:c;     x=(d>x)?x:d;
   printf("%d\n", x);
   return  0;
}
```

2.5.6　逗号运算符

C 语言中分隔符逗号“,”也是一种运算符，称为逗号运算符。其功能是把两个表达式连接起来组成一个新的表达式，称为逗号表达式。一般形式为：

表达式 1, 表达式 2, 表达式 3, …., 表达式 n;

求值过程是自左向右求各个表达式的值，并以最后一个表达式的值作为整个逗号表达式的值。逗号运算符的优先级低于赋值运算符。

【例 2.11】考查以下程序运行结果。

```
#include<stdio.h>
int  main()
{   int  a=2, b=4, c=6, x, y;
    x=a+b, y=x+c;
    printf("y=%d, x=%d", y, x);
    return  0;
}
```

本例中，“x=a+b, y=x+c;”为逗号表达式语句，运算顺序自左向右，得到 x 的值是第一个表达式的值，y 的值在 x 的基础上加上 6。最终运行结果为：

y=12, x=6

对于逗号表达式还要注意两点：

① 程序中使用逗号表达式，要注意区分使用的是逗号表达式内各表达式的值，还是整个逗号表达式的值。

② 并不是在所有出现逗号的地方都组成逗号表达式，如在变量说明中、函数参数表中的逗号只是用作各变量之间的间隔符。

【练习2.13】以下程序段的输出结果为_____。

```
#include<stdio.h>
int  main()
{
    int  x;
    printf("%d", (x=5, 2*x, 3*x));
    return  0;
}
```

2.5.7 位运算符

位运算符是对数据按二进制位进行运算的操作符。C 语言提供的位运算功能使它同时具有高级语言和汇编语言的优点，这使得 C 语言与其他高级语言相比有很大的优越性。因为一般高级语言处理的数据的最小单位只是字节，而在许多应用中经常要用到处理二进制位的运算，例如在许多系统软件中，经常对字节或字中的实际位进行检测、设置或移位，这些都可以用 C 语言中的位运算实现。

C 语言中的位运算符主要有以下几种：

~(求反)、&(按位与)、|(按位或)、^(按位异或)、>>(右移)、<<(左移)

除~是单目运算符外，其余均为双目运算符。位运算符的操作对象必须是字符型或整型，其他数据类型(如实型)不适用。所有数据在参与运算时必须转换为二进制进行运算。

1. 求反运算符~

~运算符将操作数的每个二进制位取成相反值，即 0 变 1，1 变 0。结果类型与操作数类型相同。

【例 2.12】有以下定义“unsigned i=0xd3f5, j=0; short k=0;”求~ i，~j，~k 的值。

将 i 变成二进制码为 1101 0011 1111 0101，因此

~i=0010 1100 0000 1010，即~i=0x2c09

j 的二进制码为全 0，~j 为全 1，即~j= 65535。

将 k 各位求反，~k 为全 1，因为 k 是有符号类型整数，所以~k 为-1。

2. 按位与运算符&

按位与运算作用是将两个操作数的对应位分别进行逻辑与运算。对应时采用低位与低位相对应，高位与高位相对应的原则。如 4&9 的运算过程如下：

```
      4：0100
  &   9：1001
   4&9：0000
```

很明显 4&9=0，而 4&&9=1，因此要注意位运算符&与逻辑运算符&&的区别。

根据按位与运算符&的运算规则可以实现一些特殊功能，例如可以将操作数中的指定位清零，其他位保持不变，即取操作数中的若干指定位。例如：

x=x&0177

该语句将 x 中除 7 个低二进制位外的其他位均清零。

3. 按位或运算符|

按位或运算符作用是对两个操作数的对应位分别进行逻辑或运算，对应时采用低位与低位相对应，高位与高位相对应的原则。按位或运算符经常用于将某些二进制位置 1。例如：

a=a | SET_ON

该语句将变量 a 中对应于 SET_ON 中为 1 的位置 1，其余位保持不变。

4. 按位异或^

按位异或运算符的作用是当 2 个操作数对应位不相同时，将该位置 1，否则该位置 0。

5. 移位运算符<<和>>

<<和 >>是双目运算符，如 a<<b 或者 a>>b。

移位运算的规则是将左操作数 a 的二进制值向左(<<)或向右(>>)移动由右操作数 b 指定的位数。两操作数必须为整数，且右操作数 b 为正数，也可以是值为正整数的表达式。

左移时，高位被移出(丢掉)，右边空出的低位用 0 填充；右移时，左边空出的高位的填充方式决定于右操作数的类型，如果是无符号数，则用 0 填充，如果是有符号数，某些机器用符号位填充左边空出的高位(即算术移位)，而有些机器用 0 填补(即逻辑移位)。

【例 2.13】x、y、z 变量定义如下“unsigned x=65, y=15; short z=-8;”计算下列表达式的值。

① x<<3：x=65 即 0x41，对应的二进制数为 0000000001000001，左移 3 位后变为 0000001000001000，即十进制数 520。

也可以这样理解：x<<3 等价于 x*2*2*2，即 x<<n 相当于 x 乘上 2^n。

② y>>3：y=15 即 0xf，对应的二进制数为 0000000000001111，右移 3 位后变成 0000000000000001，即十进制数 1。

也可以这样理解：y>>3 等价于 y/2/2/2 或 y/(2×2×2)，即：y>>n 相当于 y 整除 2^n。

③ z>>2：z=-8，对应的二进制数为 1111111111111000(-8 的补码)，右移 2 位后变为 1111111111111110(-2 的补码)，即十进制数-2。

【注意】对操　数进行移位运算并不改变原操　数的值。例 2.13 中的三个移位运算并不改变 x，y，z 的值，除非通过赋值运算符 x=x<<3，才能改变 x 的值。

2.6　类 型 转 换

变量的数据类型是可以转换的。转换的方法有两种，一种是自动转换，一种是强制转换。

2.6.1　自动类型转换

自动转换发生在不同数据类型的数据混合运算时，数据宽度会自动从小变大。自动转换遵循以下规则。

① char 型和 short 型参与运算时，先转换成 int 型。

② unsigned short 型转换为 unsigned 型。

③ float 型自动转换为 double 型。

④ 经过以上转换后仍存在不同的数据类型时，按照 int-unsigned-long-double 从低到高

的顺序进行转换。

⑤ 在赋值运算中，赋值号两边数据的类型不同时，赋值号右边数据的类型将转换为左边变量的类型。如果右边数据的长度比左边长时，将进行四舍五入，丢失一部分数据，这样会降低数据精度，在编译时可能会有 warning 出现。

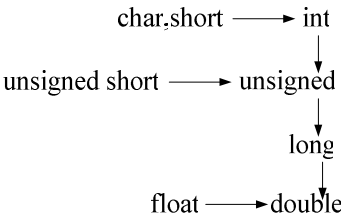


图 2.1　数据类型自动转换规则

自动类型转换的规则如图 2.1 所示。

【例 2.14】分析下面的程序段中的数据类型自动转换的过程。

```
#include<stdio.h>
int  main()
{
    char  ch='a';
    int  i=5;
    double  PI=3.14159;
    double  y;
    y= i/2 +ch+PI;
    printf("s=%f\n",y);
    return  0;
}
```

在执行“y=i/2 +ch+PI;”语句时，i 为 int 类型，i/2 结果为 2，仍为 int 型；ch 为 char 类型，先转换为 int 类型，即字符'a'的 ASCII 码，运算后的结果 99 仍为 int 型；PI 为 double 类型，因此先将前面 int 型结果 99 转换成 double 型，再进行加法运算，结果为 double 型，最后赋值给变量 y，y 也是 double 型，无需转换，结果是 double 型。

2.6.2　强制类型转换

强制类型转换是通过类型转换运算来实现的。其一般形式为：

(类型说明符)表达式

其功能是把表达式的运算结果强制转换成类型说明符所表示的类型。

例如：(float)a 把 a 转换为实型；

(int)(x+y)把 x+y 的结果转换为整型。

【注意】① 类型说明符和表达式要加括号(表达式为单个变量时可以不加括号)，如(int)(x+y)表示把 x+y 的和强制转换为 int 型，而(int)x+y 则是把 x 强制转换为 int 型之后再与 y 进行加法运算。

② 无论是强制转换或是自动转换，都只是为了本次运算需要而对数据进行的临时性转换，变量的类型并未发生改变。

【例 2.15】考查下述程序运行结果。

```
#include<stdio.h>
int  main()
{
    float  f=5.75;
    printf("(int)f=%d, f=%.2f\n", (int)f, f);
```

```
    return 0;
}
```

输出结果为：

```
(int)f=5, f=5.75
```

将 f 强制转换成 int 类型后，删去了小数部分，因此第 1 个输出是 5，但再次输出 f 的值后，结果仍为其原来的值 5.75。例 2.15 表明，f 虽强制转为 int 在运算中起作用，但这种强制转换是临时的，f 本身的类型和数值并不改变。

【练习 2.14】设有定义“double x=123.4567;”则执行以下语句后的输出结果是_____。

```
printf("%f\n", (int)(x*100+0.5)/100.0);
```

习　题　2

1. 已定义a为字符型变量，则下列语句中正确的是_____。

A. a='97'　　B. a="97";　　C. a=97;　　D. a="a";

2. 有定义语句“int x, y;”。若要通过“scanf("%d,%d", &x, &y);”语句使变量x得到数值11，变量y得到数值12，下面四组输入形式中，错误的是_____。

A. 11 12↙　　B. 11,12↙　　C. 11, 12↙　　D. 11,↙12↙

3. 以下选项中可作为C语言合法整数的是_____。

A. 10110B　　B. 0386　　C. 0Xffa　　D. x2a2

4. 有以下程序，程序运行后的输出结果是_____。

A. 6,1　　B. 2,1　　C. 6,0　　D. 2,0

```
void main()
{   int a, b, d=25;
    a=d/10%9;
    b=a&&(-1);
    printf("%d, %d\n", a, b);
}
```

5. 以下语句中，非法的赋值语句是_____。

A. n=(i=2, ++i);　　B. j++;　　C. ++(i+1);　　D. x=j>0;

6. 若函数中有以下语句“int k;”，则_____。

A. 系统将自动给 k 赋初值 0　　B. 这时 k 中的值无定义

C. 系统将自动给 k 赋初值-1　　D. k 的值为随机值

7. 以下各项中，能用作数据常量的是_____。

A. o123　　B. 028　　C. 1.3e1.2　　D. 115L

8. 有以下程序：

```
void main()
{   int k=11;
    printf("k=%d, k=%o, k=%x\n", k, k, k);
}
```

运行程序后的输出是_____。

A. k=11, k=12, k=11　　B. k=11, k=13, k=13

C. k=11, k=013, k=0xb　　D. k=11, k=13, k=b

9. 若整型变量 a，b 的值分别为 7 和 9，要求按以下格式输出 a，b 的值:

a=7

b=9

请完成输出语句：“printf("_____", a, b);”。

10. 以下各项中，不能作为 C 语言合法常量的是_____。

A. 'ab'　　B. 0.1e+6　　C. "\a"　　D. '\011'

11. 以下各项中，正确的定义语句是_____。

A. double a; b;　　B. double c=d=1;　　C. double, c,d;　　D. double c=7,d=7;

12. 若有表达式(w)?(x++):(y--)，则下述选项中可以替换式中 w 的是_____。

A. w==0　　B. w!=0　　C. w==1　　D. w!=1

13. 以下能正确定义变量且赋初值的语句是_____。

A. int n1=n2=10;　　B. char c=32;

C. float f=f+1.1;　　D. double x=12.3E2.5;

14. 以下程序的功能是给 r 输入数据后计算半径为 r 的圆面积 s，程序在编译时出错。

```
void main()
/* Beginning */
{   int r; float s;
    scanf("%d", &r);
    s=*∏*r*r;
    printf("s=%f\n", s);
}
```

出错的原因是_____。

A. 注释语句书写位置错误

B. 存放圆半径的变量 r 不应该定义为整型

C. 输出语句中格式描述符非法

D. 计算圆面积的赋值语句中使用了非法变量

15. 已知字符 A 的 ASCII 码值为 65，以下程序运行后的输出结果是_____。

```
void main()
{   char a, b;
    a='A'+'5'-'3';    b=a+'6'-'2';
    printf("%d%c\n", a, b);
}
```

16. 以下各项中，值为 1 的表达式是_____。

A. 1-'0'　　B. 1-'\0'　　C. '1'-0　　D. '\0'-'0'

第 3 章　程序控制结构

程序是若干语句的集合，依照程序中语句的执行顺序不同构成三种基本程序控制结构。简单的是顺序结构，其特征是自上而下顺序执行每一条语句；第二种是选择结构，其特征是根据某种条件是否成立，选择执行不同的语句；第三种是循环结构，其特征是根据条件(循环条件)，决定是否重复执行一些语句(循环体)。本章主要介绍这三种程序控制结构及其综合应用。

知 识 结 构

1. 顺序结构
2. 选择结构
   ① 单分支 if()结构
   ② 双分支 if()… else … 结构
   ③ 分支结构嵌套和多路分支 switch()结构
3. 循环结构
   ① for()循环
   ② while()循环
   ③ do…while()循环
   ④ 循环结构嵌套
4. 控制语句 **break** 与 **continue** 的应用
5. 控制语句的综合应用

3.1　概　　述

程序由若干语句构成，一条语句编译后，能产生若干条机器指令。每条指令就会引起一个机器系统的操作。所以程序是指令的集合。

程序应该包括数据描述(由声明部分实现)和数据操作(由语句来实现)。数据描述包括定义数据结构和在需要时对数据赋予初值。数据操作的任务是对已提供的数据进行加工。

C 语句分为以下五类。

1. 控制语句

完成一定的控制功能，C 语言中有 9 种控制语句，它们是：

① if()…else…　　(条件语句)
② for()　　(循环语句)
③ while()　　(循环语句)
④ do…while()　　(循环语句)
⑤ continue　　(结束本次循环语句)
⑥ break　　(终止执行 switch 或循环语句)
⑦ switch()　　(多分支选择结构)
⑧ goto　　(转向语句)
⑨ return　　(从函数返回语句)

2. 函数调用语句

由函数调用加一个分号构成一个语句，例如：

```
printf("hello  world1");        // 调用库函数 printf()
```

3. 表达式语句

第 2 章学习的表达式加分号就构成了表达式语句。最典型的是：由赋值表达式加分号构成一个赋值语句。例如：“i=i+1”是表达式，而“i=i+1;”是语句。可以看到，一个表达式的最后加上一个分号就成了一个语句，一个语句必须在最后出现分号，分号是语句中不可缺少的组成部分，而不是两个语句间的分隔符号，任何表达式都可以加上分号而成为语句。在 C 程序中大多数语句是表达式语句，所以有人把 C 语言称为“表达式语言”。

4. 空语句

只有分号的语句就是空语句。它什么都不做。下面就是一个空语句：

```
;
```

空语句有时用来作转向点，或循环语句中的循环体(如果循环体是空语句，表示循环体什么也不做)。

5. 复合语句

用一对大括号“{”和“}”括起来的语句就是复合语句。例如：

```
{   x=3;
    printf("x=%d", x);
}
```

一般情况下，凡是允许出现语句的地方都允许使用复合语句。在程序结构上，复合语句看做是一个整体，但是内部可能完成了一系列工作。

3.2　顺 序 结 构

所谓顺序结构，就是按照语句的先后顺序一条一条地执行。顺序控制语句是一类简单的语句，包括表达式语句、空语句和输入输出语句(或其他函数调用语句)等。程序自上而下的执行，如图 3.1 所示。

每一个步骤(step)既可以是一个语句，也可以是一个其他控制结构。

【例 3.1】从键盘输入球的半径 r，计算其体积后输出。

【分析】本题根据半径 r 求球体积，可以使用公式 $V=\frac{4\times\pi\times r^3}{3}$ 计算。其中，π 的值在运

算过程中固定不变，因此可作为常量，半径 r 根据输入的值不同，需定义为浮点型变量，通过公式计算得到的球体积 V，也是一个浮点型变量，最后将 V 输出。内存空间分配情况如图 3.2 和所示，程序流程如图 3.3 所示。

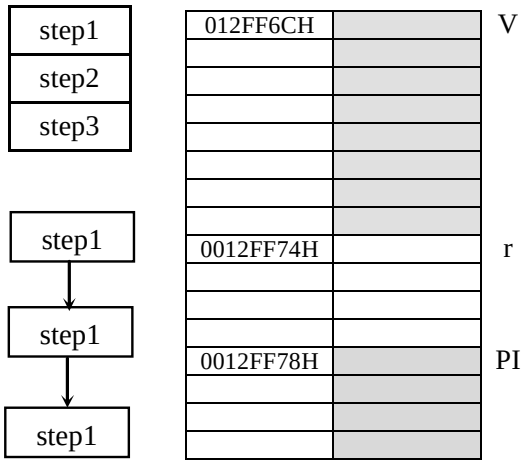


图 3.1　顺序结构图

图 3.2　变量内存空间分配图

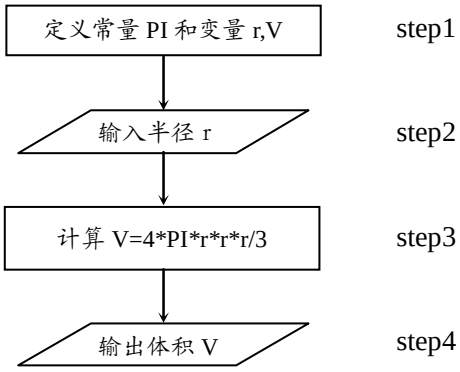


图 3.3　例 3.1 程序流程

参考代码如下：

```
#include<stdio.h>
void  main(void)
{
    //step1  定义常量和变量
    const  double  PI=3.14;
    float  r;
    double  V;

    //step2  输入半径 r
    printf("请输入半径 r：");
    scanf("%f", &r);

    //step3  计算体积 V
    V =4 * PI * r * r * r / 3;

    // step4  输出体积 V
    printf("\n 半径为 r=%.2f 的球的体积为%.2f\n", r, V);
}
```

运行实例：

```
请输入半径 r：2.0
半径为 r=2.00 的球的体积为 33.49
```

【思考】① 能否交换 step2 和 step3 两步骤的顺序？

② 对于 step1、step2、step3 到 step4，有没有能颠倒顺序后，又符合题目要求的步骤？

【　结】顺序结构　大的特征就是自上而下顺序地执行每一个语句，语句顺序颠倒后可能出现错误，也可能出现其他的运算结果。

【练习 3.1】从键盘输入两个整数，然后计算这两个数的和，最后输出结果。运行结果如下：

```
请输入两个整数：5 9
5+9=14
```

3.3 选 择 结 构

选择结构又称分支结构，它是程序的基本控制结构之一。选择结构的作用是根据给定的条件决定做什么样的操作(实现分支)，它在程序设计中被广泛使用。C 语言中有单分支结构、双分支结构和多分支结构。

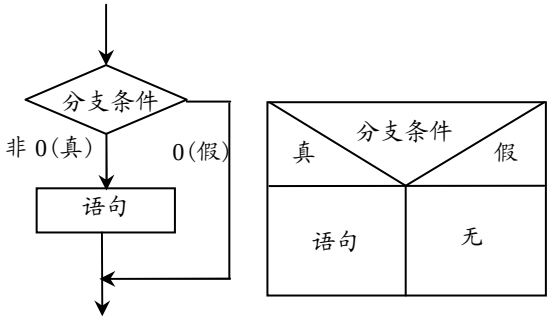
3.3.1 单分支结构

单分支结构判断给定的条件是否满足，从而决定是否执行指定的操作，其示意图如图 3.4 所示。语句格式如下：

```
if(表达式)
    语句
```

功能：计算表达式的值，如果表达式的值为真，执行后面的语句；否则不执行，即跳过该语句。例如：

```
if(a>b)
   printf("a=%d 较大。\n", a);
```



a 流程图　　b N-S 图

图 3.4 单分支结构

【说明】① 表达式可以是逻辑表达式或者关系表达式，也可以是其他表达式，甚至是一个变量判断表达式，表达式非 0 为逻辑真，0 为逻辑假。

② 这里所说的语句可以是一条语句，也可以是多条语句，甚至是一个控制结构(如果是一个控制结构，就 成了嵌套的控制结构)，如果是一条语句，不要漏掉语句后的分号；如果是多条语句，需要用“{”和“}”括起来，形成复合语句。

【例 3.2】输入两个不等的整数，输出其中的较大数。

```
#include<stdio.h>
void  main(void)
{
    int  a,b,max;
    printf("请输入两个不等的整数: ");
    scanf("%d%d", &a, &b);              // 输入两个整数
    max = a;                            // 默认较大值为 a
    if(max <b)                          // 如果 max 小于 b，则将 b 赋予 max
       max = b;
    printf("%d 与%d 中，max=%d.\n", a, b, max);
}
```

运行实例：

请输入两个不等的整数：22 36

22 与 36 中，max=36.

【说明】本例中首先输入两个整型数据，分别赋予变量 a 和 b；再将 a 的值赋予变量 max，然后用 if 语句判断 max 和 b 的关系，如果 b 较大，则将 b 赋予 max。因此，max 中　是保持一个较大的数。

【练习 3.2】输入一个整数，用单分支结构求其绝对值，并输出。

3.3.2　双分支结构

根据给定的条件，从两组操作中选择一组执行，即双分支结构，其示意图如 3.5 所示。语句格式如下：

```
if(表达式)
    语句 1
else
    语句 2
```

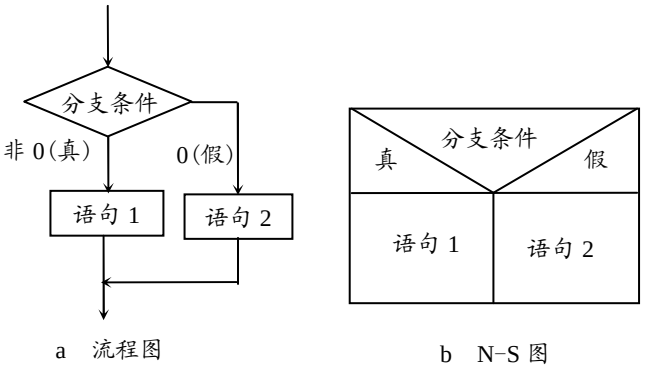


图 3.5　双分支结构

功能：如果表达式的值为逻辑真，则执行语句 1，否则执行语句 2。例如：

```
if(a>b)
    printf("max=%d\n", a);
else
    printf("max=%d\n", b);
```

【说明】① 在双分支语句中，else 必须与 if 配对使用，构成 if-else 语句，实现双分支选择。如果没有 else，即成为单分支结构。

② 语句 1 和语句 2，既可以是一个语句，也可以是多个语句甚至是一个控制结构，若是多个语句需要用“{”和“}”括起来，形成复合语句。

【例 3.3】输入 3 个整数，输出最大数。

【分析】定义 3 个 int 型变量，用来存储键盘输入的 3 个整数，比较 a 和 b，若 a 大，则把 a 赋值给 max，否则把 b 赋值给 max；然后再用 max 和 c 比较，若 c 大，把 c 赋给 max，最后 max 则为 a、b、c 中最大的值。流程图如图 3.6 所示。

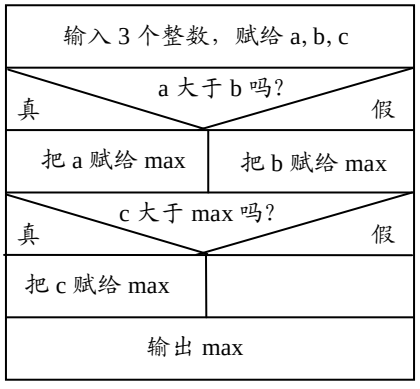


图 3.6　求 a、b、c 中最大值的 N-S 图

参考代码如下：

```
#include<stdio.h>
void  main(void)
{
    int  a, b, c, max;
    printf("请输入三个整数：");
    scanf("%d%d%d",&a,&b,&c);
    /* 比较 a、b 大小，把较大的赋给 max */
    if (a>b)          //比较 a 和 b 的大小
        max=a;
    else
        max=b;
```

```
        /* a、b 中的较大者 max 与 c 再比较大小 */
        if(c>max)      // 如果 c 大于 a 和 b 中的较大者 max ，把 c 赋给 max
            max=c;

        //输出结果
        printf("%d, %d, %d 中，最大值 max=%d.\n", a, b, c, max);
    }
```

运行实例：

```
请输入三个整数：22  36  6
22, 36, 6 中，最大值 max=36.
```

【说明】首先输入 3 个整型数据，分别赋给变量 a、b 和 c；先用 if 语句，判断 a>b 表达式的值，若 a 大，则把 a 赋给 max，否则把 b 赋给 max，即把 a 和 b 中较大的数赋予变量 max；然后用 if 语句判断 max 和 c 的关系，如果 c 较大，则将 c 赋予 max。 终 max 中是三数中大的数。

【练习 3.3】输入一个整数，判断并输出它是偶数还是奇数。

【练习 3.4】一个少年乘坐公交车，如果身高超过了 1.4m，则应按成人投票 1 元，否则投 0.5 元。现在输入身高，输出应投票的金额。

3.3.3 分支结构的嵌套

分支结构的语句 1 和语句 2 既可以是一个简单的语句，也可以是另一个分支结构。例如：

```
if(表达式 1)
    if(表达式 2)
        语句 1              //表达式 1 为真，且表达式 2 也为真时执行。
    else
        语句 2              //表达式 1 为真，且表达式 2 为假时执行。
else
    语句 3                  //表达式 1 为假时执行。
```

同理也有如下结构：

```
if(表达式 1)
    语句 1                  //表达式 1 为真时执行。
else
    if(表达式 2)
        语句 2              //表达式 1 为假，且表达式 2 也为真时执行。
    else
        语句 3              //表达式 1 为假，且表达式 2 为假时执行。
```

```
if(表达式 1)
    if(表达式 2)
        语句 1              //表达式 1 为真，且表达式 2 也为真时执行。
    else
        语句 2              //表达式 1 为真，且表达式 2 为假时执行。
else
    if(表达式 3)
        语句 3              //表达式 1 为假，且表达式 3 也为真时执行。
    else
        语句 4              //表达式 1 为假，且表达式 3 为假时执行。
```

其流程图分别如图 3.7 中的(a)、(b)、(c)所示，嵌套 N-S 流程图如图 3.8 的(a)、(b)、(c)所示。

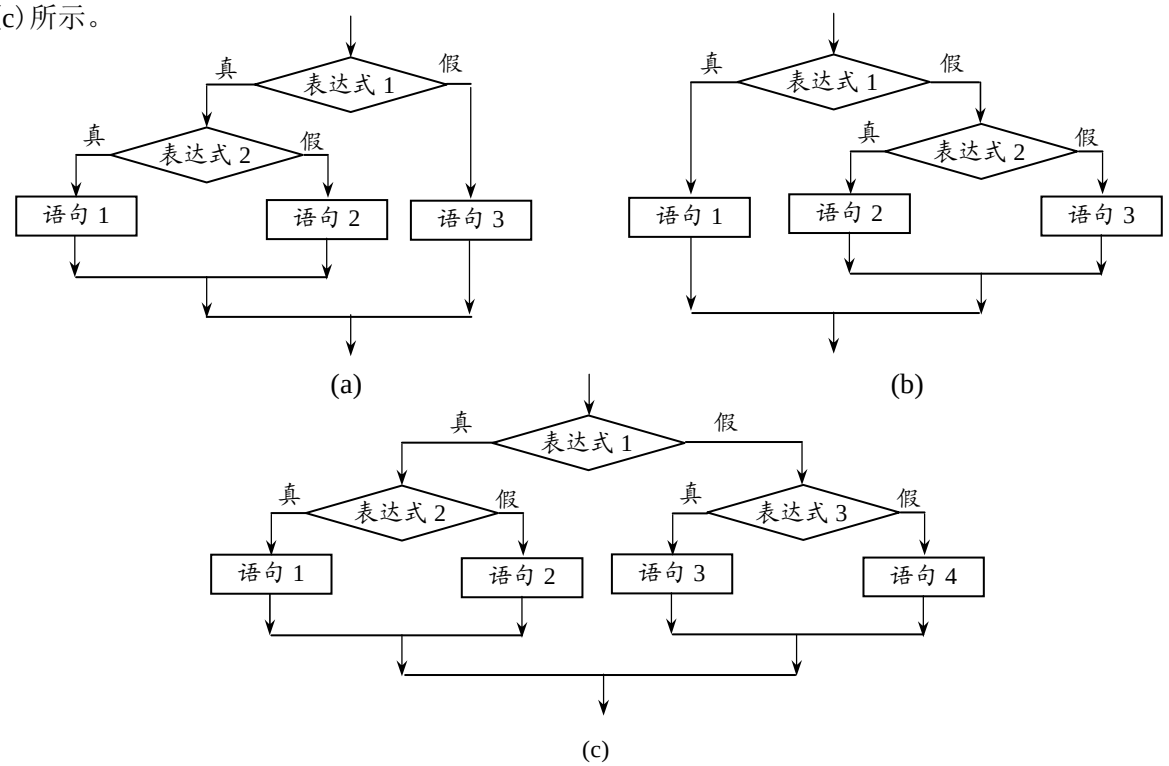


图 3.7　分支结构嵌套的流程图

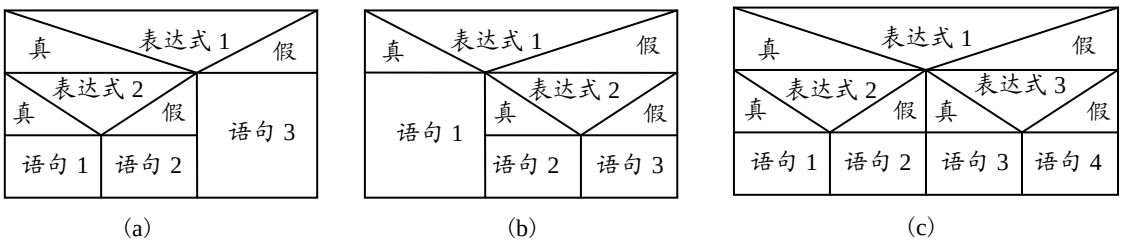


图 3.8　分支结构嵌套的 N-S 图

① 分支结构的嵌套不仅可以嵌套双分支结构，还可以嵌套单分支结构。例如：

```
if(表达式 1)
    语句 1
else
    if(表达式 2)
        语句 2
```

② else 分支内嵌套分支结构时，可以把 if 与 else 写在同一行。例如：

```
if(表达式 1)
    语句 1
else  if(表达式 2)
    语句 2
```

上述①和②的结构形式虽然不同，但是含义相同。①涉及的结构化更强，但是若情况较多时，需要逐步缩进，此时②所涉及的结构就发挥了优势(可参考多路分支部分的叙述)。

③ 分支结构嵌套时，注意 else 与 if 的匹配，首先遵循的原则是，else 总是与其前面最近的未匹配的 if 匹配，构成一个双分支结构。

例如：

```
if(表达式 1)
    if(表达式 2)
        语句 1
    else
        语句 2
```

【思考】上述语句 2 的执行条件是什么？

【提示】else 与表达式 2 对应的 if 匹配。所以语句 2 执行的条件是表达式 1 为真，表达式 2 为假。

【练习 3.5】执行下面的程序后，变量 y 的值是_____。

A. 1000　　B. 1200　　C. 1400　　D. 3400

```
#include<stdio.h>
void  main()
{
    int  x, y;
    x=7;
    if(x<6)
            y=1000;
    else
            if(x<=18)
                y=(x-6)*200+1000;
            else
                y=(x-18)*300+3400;
    printf("y=%d.\n", y);
}
```

3.3.4　多路分支结构

根据一个条件的真假，可以分成两支解决问题的路线。若一个问题有 n 种情况，就需要 n−1 个条件。例如：一个问题有 3 种情况，则需要 2 个条件进行判断后，构成分支结构。我们称依次判断给定的 n 个条件，以确定从 n+1 组操作中选择某一组的结构称为多分支结构。根据以上的学习，用嵌套的分支结构能够解决一个多种分支的问题，如图 3.9 所示。

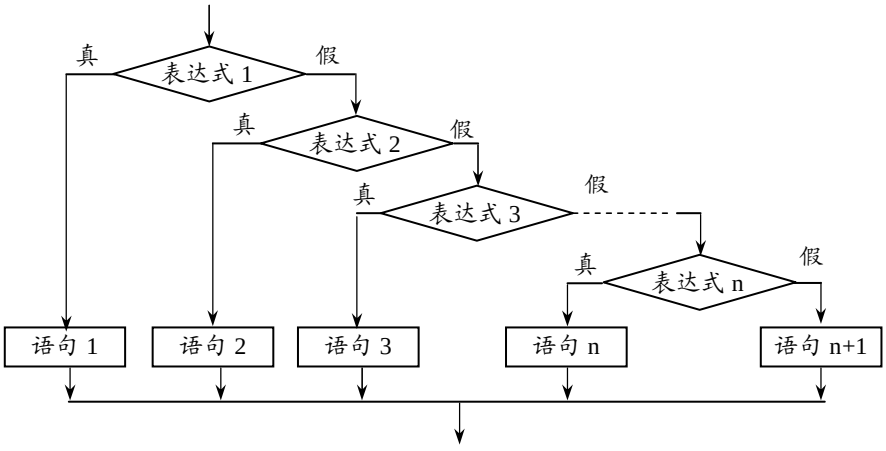


图 3.9　多路分支结构流程图

1. 采用分支嵌套形式实现多分支结构

若情况较多时，多分支结构可以用 else if 结构实现。else if 结构的一般格式：

```
if(表达式 1)
    语句 1
else if(表达式 2)
    语句 2
else if(表达式 3)
    语句 3
……
else if(表达式 n)
    语句 n
else
    语句 n+1
```

或者写成下述分支嵌套形式

```
if(表达式 1)
    语句 1
else
    if(表达式 2)
        语句 2
    else
            if(表达式 3)
              语句 3
```

```
……
else
    if(表达式 n)
       语句 n
    else
       语句 n+1
```

上述两种形式的区别：前者没有阶梯缩进，后者阶梯缩进，前者使用语句 else if 结构，后者采取 else 中嵌套分支结构。另外在条件为真的语句中也可以是嵌套的选择结构。

【注意】当需要判断的情况较多时，建议使用 else if 结构；情况较少时，使用阶梯化的分支嵌套结构。

【例 3.4】根据字符的 ASCII 值，判断键盘输入的一个字符是数字、大写字母字符，还是小写字母，并输出说明结果。

【分析】根据输入的字符，逐步判断字符所在的范围，设计步骤如下：

① 先输入一个字符到字符变量 c；

② 判断 c 是否在字符 '0' 和字符 '9' 之间，如果是，输出“这是一个数字字符”的结果，并跳到步骤⑥，否则进入步骤③；

③ 判断 c 是否在字符 'A' 和字符 'Z' 之间，如果是，输出“这是一个大写字符”的结果，并跳到步骤⑥，否则进入步骤④；

④ 判断 c 是否在字符 'a' 和字符 'z' 之间，如果是，输出“这是一个小写字符”的结果，并跳到步骤⑥，否则进入步骤⑤；

⑤ 输出“这是一个其他字符”；

⑥ 结束。

参考代码：

```
#include<stdio.h>
void  main(void)
{
    char  c;
    printf("请输入一个字符：");
    c = getchar();
    if(c >='0' && c <= '9')
        printf("这是一个数字字符。\n");
    else  if(c >= 'A' && c <='Z' )
        printf("这是一个大写字母。\n");
    else  if(c >= 'a' && c <='z')
        printf("这是一个小写字母。\n");
    else
        printf("这是一个其他字符。\n");
}
```

运行实例 1：

```
请输入一个字符：W
这是一个大写字母。
```

运行实例 2：

```
请输入一个字符：e
这是一个小写字母。
```

运行实例 3：

请输入一个字符：2
这是一个数字字符。

运行实例 4：

请输入一个字符：#
这是一个其他字符。

【例 3.5】为鼓励节约用水，根据用水量采取阶梯价格策略，即当月用水量低于 $10m^3$ 时，水价为 2.5 元/m^3，当月用水量超过 $10m^3$ 时，超过的部分水价为 5 元/m^3。请根据输入的某月用水量(单位：m^3)，输出应付的水费。

【分析】首先应该输入用水量(可以设为 w)，如果有 0≤w≤10，则该月水费(设为 m)为 2.5×w，否则，若 w>10，则该月水费为(w−10)×5+10×2.5，其他为非法输入数据。根据题意可以画出流程图，如图 3.10 所示。

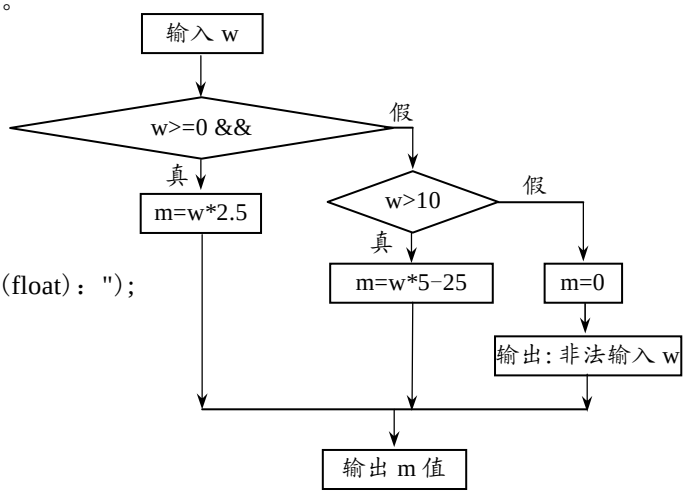


图 3.10　流程图

参考代码：

```
#include<stdio.h>
void  main(void)
{
    float  w;
    double  m;
    printf("请输入一个用水量 w(float)： ");
    scanf("%f", &w);
    if(w>=0 && w<=10)
        m = w * 2.5;
    else
        if(w>10)
            m = w * 5 - 25;
        else
        {
            m=0;
            printf("非法输入 w。\n");
        }
     printf("水费 m=%.2f\n", m);
}
```

运行实例 1：

请输入用水量 w(float)： 6.66
水费 m=16.65

运行实例 2：

请输入用水量 w(float)： 18.50
水费 m=67.50

运行实例 3：

请输入用水量 w(float)： -20
非法输入 w
水费 m=0.00

【说明】else 与上面同一层次的　近未匹配的 if 构成为一对 if…else 结构。

【练习 3.6】某公司销售人员的工资算法如表 3.1 所示。销售的产品单价为 1000 元/件，

若每月销售 5 件以下(不含 5)，没有提成；5 件以上 10 件以下(含 5，不含 10)，该部分每件提成 4%；若每月销售 10 件以上 15 件以下(含 10，不含 15)，该部分每件提成 6%；每月销售 15 件以上(含 15)，该部分每件提成 10%。编制程序完成以下要求：输入销售件数，输出销售人员的工资。

表 3.1　销售提成计算表

销售件数 x	底薪工资(元)	提成比例
[1,5)	2000	0%
[5,10)	2000	4%
[10,15)	2200	6%
15 以上(含 15)	2500	10%

【练习 3.7】某市出租车的收费情况只有两种。一种是起步价为 7 元，车程 3 公里；3 公里以上每公里 1.2 元；6 公里以上要加收 50%的回空费，即每公里 1.8 元。另一种是排气量在 1.8 升以上的豪华型轿车，起步价 10 元，车程也是 3 公里；3 公里以上每公里 1.8；6 公里以上加收 50%的回空费，即每公里 2.7 元。低速行驶及等待，每 5 分钟按照 1 公里计费。编制程序完成以下要求，输入乘坐的车型、行驶里程数和低速行驶时长，计算并输出应付的出租车费(四舍五入到元)。

2. 采用 switch 语句实现多分支结构

C 语言提供了另一种 switch 语句实现多分支结构。其一般格式为：

```
switch(表达式)
{
  case 常量表达式 1: 语句 1
  case 常量表达式 2: 语句 2
  ……
  case 常量表达式 n: 语句 n
  default: 语句 n+1
}
```

功能：先计算表达式的值，然后逐个与 case 中常量表达式的值比较，如果二者相等，执行相应 case 后面的语句；不相等时，继续执行下一个 case 结构进行判断。一旦相等(即匹配成功)，不再执行比较，继续执行后面所有 case 后的语句，直到有 break 终止该结构。若表达式的值与所有 case 中的表达式的值均不相等时，执行 default 后面的语句。

【说明】① 小括号中表达式的类型必须是整型或字符型。

② case 后面的“常量表达式”的值必须是整型或字符型。

③ case 后面的常量表达式的值互不相同，否则会出现矛盾的现象。

④ case 后面的“常量表达式”起一个程序入口的标号　用。系统一旦找到入口标号，就从此标号处开始执行，不再进行符号判断。所以为了终止一个分支的执行，需要在相应的分支末尾加一个 break 语句。

⑤ break 语句的　用是终止当前执行的语句。在这里是跳出 switch 语句，使得程序转向 switch 后面的语句。

⑥ 多个 case 语句可以共同使用一　执行语句。

⑦ switch 语句的执行部分用一对“{”、“}”括起来，不能省略，default 子句可以省略。

⑧ switch 语句可以嵌套使用。

【例 3.6】输入学生成绩等级，输出其对应的分数段。

【分析】本题是通过输入一个考试成绩的等级，调用 scanf()函数，使用%c 的输入格式获得键盘输入值，通过判断等级字符的不同，输出不同的结果。流程图如图 3.11 所示。

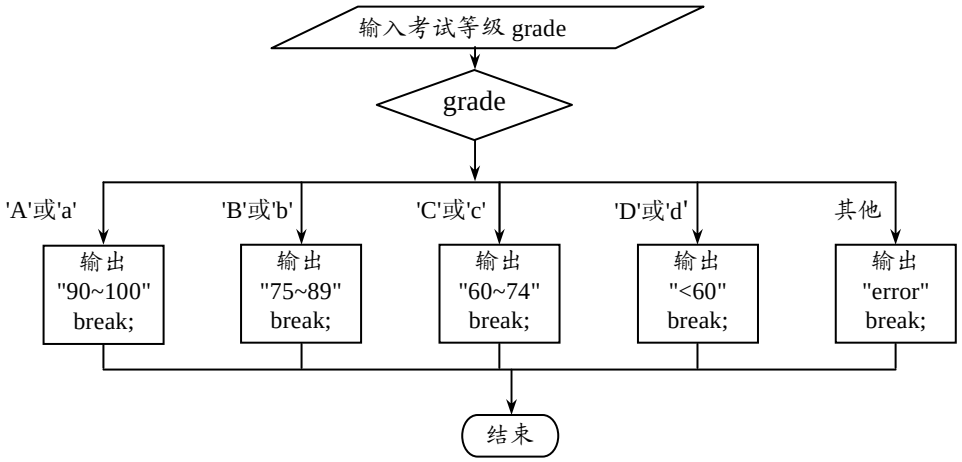


图 3.11　程序流程图

参考代码：

```
#include<stdio.h>
void  main(void)
{
    char  grade;
    printf("请输入成绩的等级：");
    scanf("%c", &grade);
    switch(grade)
    {
        case  'A':
        case  'a': printf("90~100\n");    break;
        case  'B':
        case  'b': printf("75~89\n");     break;
        case  'C':
        case  'c': printf("60~74\n");     break;
        case  'D':
        case  'd': printf("<60\n");        break;
        default: printf("Input  error.\n");
          }
}
```

运行实例 1：

```
请输入成绩的等级：A
90~100
```

运行实例 2：

```
请输入成绩的等级：c
60~74
```

运行实例 3：

```
请输入成绩的等级：v
Input  error.
```

【思考】如果在 case 分支后面省去 break 语句，结果会有什么不同吗？

【练习 3.8】用 switch 结构编制程序，输入 1 个学生的 C 语言课程考试成绩(百分制整型

数)，输出成绩相应等级。

A 级：90～100　B 级：80～89　C 级：70～79　D 级：60～69　E 级：<60

【练习 3.9】输入年月，输出该月有几天。

运行实例：

请输入年月：2010-3

2010 年 3 月有 31 天。

【练习 3.10】用 switch 语句编制程序，输入 10 个学生的 C 语言成绩(百分制整型数)，统计各等级的人数。

A 级：90～100　B 级：80～89　C 级：70～79　D 级：60～69　E 级：<60

3.4　循 环 结 构

在生活中我们经常遇到重复的过程，例如重复地做加法运算、乘法运算或者重复地进行一个其他操作等，这就需要用到循环结构。循环结构是 C 语言程序的三种结构之一，其作用是在一定条件下重复执行一组操作，从而把复杂的、不易理解的求解过程转化为易于理解的、操作简单的多次重复过程。

本节将介绍三种不同的循环语句，即 for 语句、while 语句、do-while 语句，可以分为当型循环和直到型循环两种形式，其基本结构如图 3.12 和图 3.13 所示。

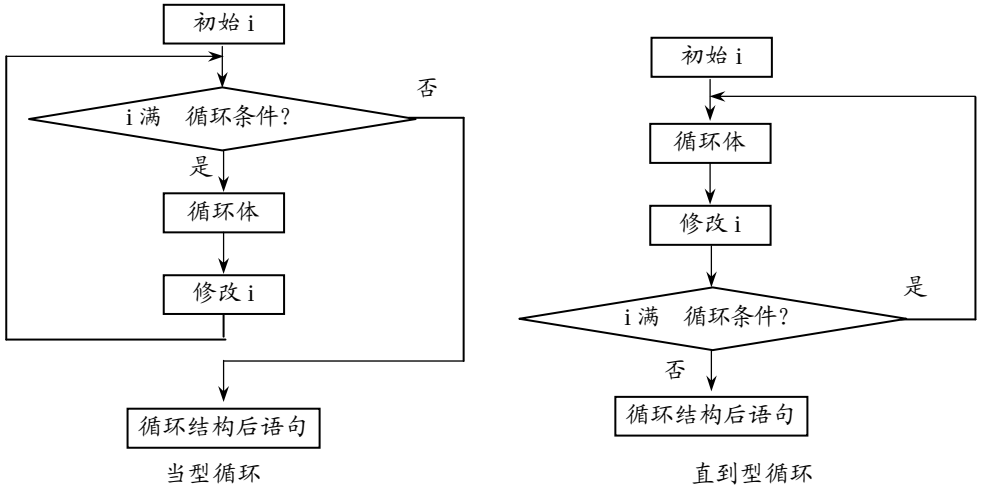


图 3.12　循环结构流程图

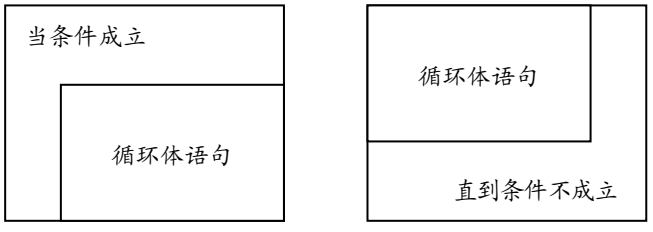


图 3.13　循环结构 N-S 图

3.4.1　for 循环结构

已知循环次数时，往往使用 for 循环结构，for 循环结构是当型循环的一种，其一般格式为：

```
for(表达式 1; 表达式 2; 表达式 3)
    循环体语句
```

执行过程如下：

① 求解表达式 1 的值；

② 求解表达式 2 的值，若其值为“假”(值为 0)，则结束循环转移到第④步。若其值为“真”(值为非 0)，则执行 for 语句内嵌的循环体语句；

③ 求解表达式 3，返回第②步；

④ 执行 for 语句的后续语句。

【例 3.7】计算 1+2+3+…+99+100 的值。

【分析】从 1 到 100 之间(含端点 1 和 100)重复做累加运算，步骤如下：

① 定义整型累加和变量 sum，整型计数器变量 i，并分别进行初始化：sum=0 和 i=1；

② 判断 i 是否不大于 100，若为“真”，顺序执行步骤③，否则跳到步骤⑤；

③ 计算累加和 sum=sum+i；

④ i 增加 1，即计算 i=i+1，然后跳到步骤②；

⑤ 执行 for 循环结构后的语句。

其 N-S 图如图 3.14 所示。

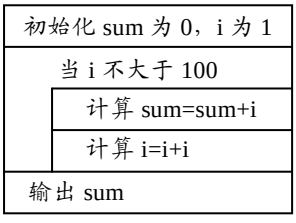


图 3.14　for 循环 N-S 图

参考代码：

```
#include<stdio.h>
void  main()
{
    int  i, sum=0;
    for(i=1; i<=100; i++)
        sum=sum+i;
    printf("1+2+…+99+100=%d", sum);
}
```

【说明】① 表达式 1：for 循环结构初始化语句，先执行，且只执行一次。可以是逗号表达式，主要用来给循环体执行前的变量赋初始值；某些情况下它可以省略，但其后面的分号不能省略。例如下面两部分代码运行效果等价：

```
i=sum=0;                          for(i=0, sum=0; i<100; i++)
for( ; i<100; i++)                    sum=sum+i;
    sum=sum+i;
```

② 表达式 2：判断表达式，判断是否继续循环，表达式 2 也可以省略，但其后的分号不能省略。当省略表达式 2 时，系统不再判断循环条件，默认其值始终为“真”，此时在循环体内应该有一个判断是否退出循环的语句。例如下面两部分代码中，前者是个死循环，后者具备退出循环的条件(break 语句可以参考 3.5 节)。

```
for(i=sum=0; ;i++)
    sum=sum+i;
```

```
for(i=1, sum=0;   ;i++)
{
    sum=sum+i;
    if(i>=100)
        break;
}
```

③ 表达式 3: 每次循环后执行，往往用来实现修改循环控制变量的功能。它也可以省略。因每次执行循环体后，要执行该表达式，所以可以将其放在循环体后， 为循环体的一部分。

【例 3.8】用 for 循环结构编程求 1 到 100(含端点 1 和 100)中所有偶数的累加和。

【分析】从 1 到 100 之间作所有偶数累加运算，题意与上一例题有一点区别，就是累加的不是所有的数，要看这个数是否为偶数，所以完成 1 到 100 的，所有偶数累加求和，步骤如下：

① 定义整型累加和变量 sum，整型计数器变量 i，并分别初始化 sum=0 和 i=1；

② 判断 i 是否小于等于 100，若为“真”，顺序执行步骤③，否则跳到步骤⑤；

③ 判断 i 是否为偶数，若是，则计算累加和 sum=sum+i；

④ i 增加 1，即计算 i=i+1，然后跳到步骤②；

⑤ 执行 for 循环结构后的语句。

N-S 图如图 3.15 所示。

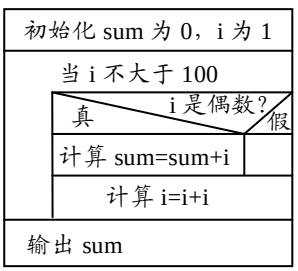


图 3.15 **for** 循环 N-S 图

参考代码：

```
#include<stdio.h>
void  main()
{
    int  i, sum=0;
    for(i=1; i<=100; i++)
      if(i%2 == 0 )
          sum=sum+i;
    printf("1 到 100 之间所有偶数的累加和为：%d。\n", sum);
}
```

【说明】① 上述程序中的循环体是一个单分支结构，当 i 不为偶数时，不进行累加，相当于执行空语句。

② 例 3.8 也可以用改变循环控制变量变化长度的方法完成，即让 i 从 2 开始，每次加 2，则每次的 i 值都是偶数，就不用再判断 i 是否为偶数，直接累加就行了，循环结束条件 i<=100 可以不变，部分代码如下(注意加粗显示的内容)：

```
int  i, sum=0;
for(i=2; i<=100; i=i+2)
    sum=sum+i;
```

【 结】① 应用 for 循环结构，要具备明确的循环开始和结束条件。

② 对循环控制变量要明确步长，要使得循环控制变量逐渐趋向于循环结束，否则将得到

错误的循环结构——死循环。

【练习 3.11】用 for 循环结构编程求 1+3+5+…+97+99 的值(可以采取两种方法完成)。

3.4.2　while 循环结构

用 while 语句构成的循环也是一种当型循环，其一般格式为：

```
while(表达式)
        循环体语句
```

当表达式为真时，执行循环体，否则结束循环，所以说，循环体可能一次都没有执行，就结束了循环。

while 语句的执行过程如下：

① 判断表达式，如果表达式为真，顺序执行步骤②，否则转到步骤③；

② 执行循环体语句，执行后跳至步骤①；

③ 执行 while 语句后面的语句。

【例 3.9】用 while 循环结构编程求 1+2+3+……+99+100 的值。

【分析】具体求解的步骤有：

① 初始化变量，让 i=1，sum=0；

② 判断表达式 i<=100，若为真，执行步骤③，否则执行步骤④；

③ 计算累加和 sum=sum+i 和控制变量 i=i+1，计算结束后跳到步骤②；

④ while 循环结构后面的语句。

N-S 图如图 3.14 所示。

参考代码：

```
#include<stdio.h>
void  main()
{
    int  i=1, sum=0;
    while(i<=100)
    {
            sum=sum+i;
            i=i+1;
    }
    printf("1+2+…+99+100=%d\n", sum);
}
```

【说明】① 每次执行循环前判断 while 后面括号内的表达式，若为真(非 0)则进入循环，否则结束循环。该表达式可以是“永真”表达式，例如 while(1)，则每次都进入循环，执行循环体，这时在循环体内往往有结束循环的语句。例如下面程序段也可以求 1 到 100 的整数和。

```
int  sum=0; i=1;
while(1)
{
   sum=sum+i
   i++;
   if(i>100)
```

```
            break;      // break 语句是一种控制语句，此处的  用是退出循环
    }
```

② 循环体语句可以是一条语句，也可以是多条语句，若是多条语句必须用“{”和“}”括起来，构成复合语句。

③ 循环体内要有能够使得循环趋于结束的语句。

④ 与 for 循环结构相比，while 循环结构是将 for 循环结构的表达式 1 放在循环结构前，表达式 3 放在循环体内的一种变形形式，因此一般的 for 循环结构都可以用 while 循环结构实现，同样 while 循环结构的程序也可以用 for 循环结构实现。

【练习 3.12】用 while 循环结构编程求 1+3+5+…+97+99 的值。

【练习 3.13】用 while 循环结构编程求 $1+\frac{1}{3}+\frac{1}{5}+\cdots+\frac{1}{97}+\frac{1}{99}$ 的值。

【练习 3.14】根据公式 $\frac{\pi}{4}=1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\frac{1}{9}-\frac{1}{11}+\cdots$，求 π 的近似值，要求最后一项的绝对值小于 1e-7。

3.4.3 do-while 循环结构

do-while 循环结构是直到型循环，它首先执行循环体，然后再判断是否满足继续循环的条件，其一般格式是：

```
do
{
    循环体语句
}while(表达式);
```

do-while 循环执行过程如下：

① 执行循环体语句；

② 判断表达式，若表达式为真(非 0)，则跳到步骤①，否则执行步骤③；

③ 执行 do-while 循环结构后面语句。

【说明】① do-while 循环结构因先执行循环体语句，再判断是否继续进行循环，所以该循环体语句至少执行 1 次。

② C 语言的 do-while 循环结构，直到循环条件为假退出循环，这跟其他编程语言(如：pascal、Bisic 等)的直到型循环结构是不同的。

③ 为了使得表达式逐步趋向于循环结束，循环体内要有修改循环控制变量的语句。

④ do-while 循环结构中，无论循环体语句有几个，都需要用“{”和“}括起来。

⑤ do-while 结构中，“while(表达式)”后面的分号不能省略。

【例 3.10】用 do-while 循环结构编程，求 1+2+3+……+99+100 的值。

【分析】求解的步骤如下：

① 初始化变量：i=1，sum=0；

② 计算累加和 sum=sum+i 和控制变量 i=i+1；

③ 判断表达式 i<=100，若为真，执行步骤②，否则执行步骤④；

图 3.16 **do-while 循环 N-S 图**

④ 执行 do-while 循环结构后面的语句。

N-S 图如图 3.16 所示。

参考代码：

```
#include<stdio.h>
void  main()
{    int  i=1, sum=0;
     do
     {
              sum=sum+i;
              i=i+1;
     }while(i<=100);
     printf("1+2+…+99+100=%d\n", sum);
}
```

【思考】① 循环体内两条语句顺序能够颠倒吗？颠倒了有什么影响？

② 若颠倒循环体内两条语句，如何修改原程序，才能符合题意？

【例 3.11】编程序完成要求：输一个正整数，然后逆序输出该数，并输出这是个几位数。

【分析】解题步骤为：

① 输入 int 型变量 num 的值，将计数器 cnt 初始化为零；

② 取 num 的最后一位数并输出，即输出 num%10 的结果；

③ 计数器 cnt++；

④ 通过使 num=num/10，去掉 num 的最后一位数；

⑤ 若 num 不为 0，则返回步骤②。

参考代码：

```
#include<stdio.h>
void  main(void)
{
    int  num, cnt=0;
    printf("请输入一个正整数：");
    scanf("%d", &num);
    printf("逆序输出：");
    do
    {
          printf("%d ", num%10);
          cnt++;
          num=num / 10;
    }while(num!=0);
    printf("共有%d 位。\n", cnt);
}
```

运行实例：

```
请输入一个正整数：268
逆序输出：8 6 2  共有 3 位。
```

【练习 3.15】用 do-while 结构，计算表达式 1+3+5+7+9+…97+99 的值。

【练习 3.16】根据公式 $\frac{\pi}{4}=1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\frac{1}{9}-\frac{1}{11}+\cdots$，用 do-while 结构求 π 的近似值，要求直到最后一项的绝对值不小于 1e-7。

【练习 3.17】根据 s=1+2+4+7+11+…+x，(x<100)，计算 s 的值。

【提示】从 1 开始　累加，累加对象 i 与前一项的差值为 1，2，3……越来越大。

3.4.4　循环结构的嵌套

循环体既可以是一条语句，也可以是多条语句，还可以是一个控制结构。当循环体本身也需要一个循环结构时，就构成循环结构的嵌套情况，如表 3.2 所示。

表 3.2　几种循环嵌套结构

1. for 循环嵌套 for 循环	2. for 循环嵌套 while 循环	3. for 循环嵌套 do-while 循环

for(表达式1; 表达式2; 表达式3)
{
…
for(表达式 4; 表达式 5; 表达式 6)
{
内循环体
}
…
}
	for(表达式 1; 表达式 2; 表达式 3)
	{
	…
	while(表达式 4)
	{
	内循环体
	}
	…
	}
		for(表达式 1; 表达式 2; 表达式 3)
		{
		…
		do
		{
		内循环体
		}while(表达式 4);
		…
		}

【说明】① 因为三种循环结构的每一种都可以被另一种循环结构嵌套，所以除以上三种嵌套情况外，还有六种嵌套的具体情况。

② 当外循环是 for 循环结构或者 while 循环结构，且循环体只嵌有一个循环结构时，可以省略外循环的大括号“{”和“}”。

【例 3.12】编制程序，分 9 行 9 列显示九九乘法口诀表。

【分析】让变量 i 从 1 变到 9，作为每一行的第一个乘数，每个 i 的值分别和 1 到 9 相乘，并显示乘积。程序如下，运行结果如图 3.17 所示。

```
#include<stdio.h>
void  main()
{
    int  i, j;
    for(i=1; i<=9; i++)                          //i 为外循环控制变量
    {                                            //外循环体开始
        for(j=1; j<=9; j++)                      //j 为内循环控制变量
        {                                        //内循环体开始
            printf("%d*%d=%2d ", i, j, i*j);          //共执行 81 次
        }                                        //内循环体结束
        printf("\n");                            //共 9 次换行
    }                                            //外循环体结束
}
```

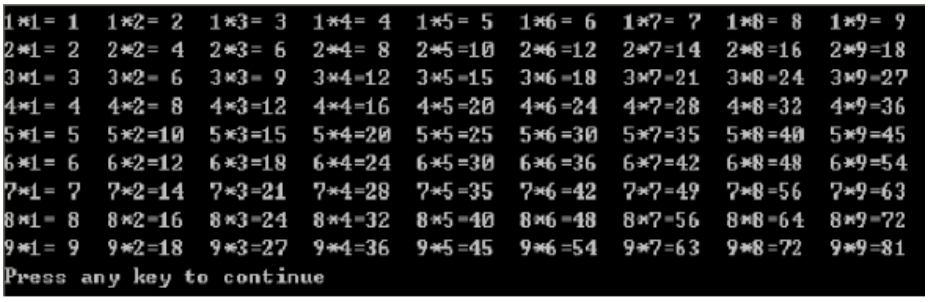


图 3.17　例 3.12 运行结果

【思考】如何修改上述程序，得到如图 3.18 所示的显示形式？

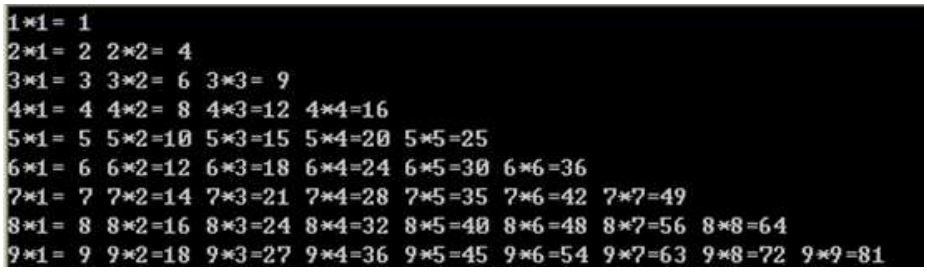


图 3.18　九九乘法表

【提示】图 3.18 中，第 1 行显示 1 个口诀，第 2 行显示两个口诀……第 9 行显示 9 个口诀，为实现这一点，只要把内循环中的 j<=9 改为 j<=i 即可。

【练习 3.18】输入整数 n，输出 n 行星号，第 1 行 1 个，第 2 行 3 个，第 3 行 5 个……例如输入正整数 3。

输出如下图形：

```
    *
  * * *
* * * * *
```

【练习 3.19】编制程序，求 1!+2!+3!+…+n!的值，n 为键盘输入的正整数。

3.5　break 和 continue 控制语句

3.5.1　break 语句

break 语句是终止语句，用在循环结构和 switch 结构中，作用是使得程序退出当前层次的循环结构或当前层次的 switch 选择结构。

【例 3.13】下面程序在执行到 i 大于 5 时，将会退出 for 循环。

```
#include<stdio.h>
void  main()
{
    int  i;
    for(i=0;i<10;i++)
    {
```

```
            if(i>5)
                break;
            else
                printf("%d", i);
        }
    }
```

运行实例：

```
0 1 2 3 4 5
```

【说明】break 语句功能是终止该层的循环结构，当 i 大于 5 时，将执行 break 语句。

【例 3.14】输入一个 n，输出紧随 n 后面能够被 3 整除的一个数。

【分析】从 n+1 开始循环，找到第一个能够整除 3 的整数后，退出程序。

参考代码：

```
#include<stdio.h>
void  main()
{
    int  i, n;
    printf("请输入一个正整数：");
    scanf("%d", &n);
    for(n=n+1; ;n++)
        if(n % 3 == 0)
        {
            printf("%d 能够被 3 整除。\n", n);
            break;
        }
}
```

运行实例：

```
请输入一个正整数：5
6 能够被 3 整除。
```

【练习 3.20】输入一个正整数 n，输出紧靠 n 前的能够被 3 或 5 整除的整数。

【练习 3.21】下面程序执行后，i 的值是_____。

```
#include <stdio.h>
void  main()
{
    int  i;
    for(i=0;i<10;i++)
        if(i%3!=0)
            printf("%d ",i);
        else
            break;
    printf("\ni=%d",i);
}
```

3.5.2　continue 语句

continue 语句往往结合分支结构用在循环结构中，作用是跳过本次循环进入下一次循环。例如语句结构：

```
for(表达式 1; 表达式 2; 表达式 3)
    if(条件表达式)
        continue;
    else
        …;
```

表示当条件表达式为真时，continue 语句执行，即跳过之后的循环体语句，直接执行表达式 3，然后执行表达式 2。

【例 3.15】计算 1 到 100 的不能被 5 整除的整数和。

【分析】i 从 1 到 100 进行循环，若 i 能够整除 5，跳过本次循环，进入下一次循环，否则(即不能整除 5)，就进行求和运算。

流程图和 N-S 图如图 3.19 和 3.20 所示。

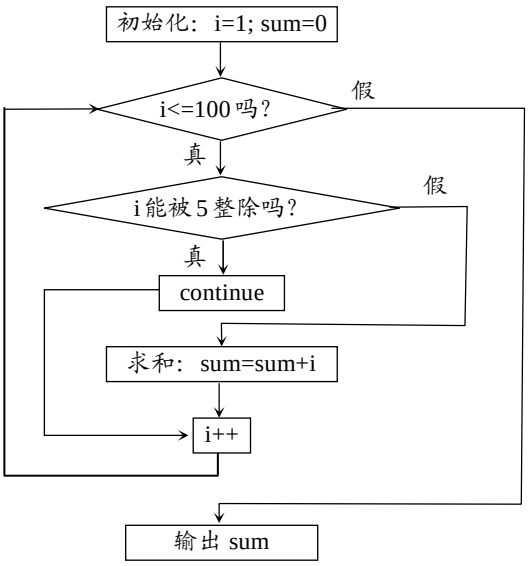


图 3.19　例 3.15 程序流程图

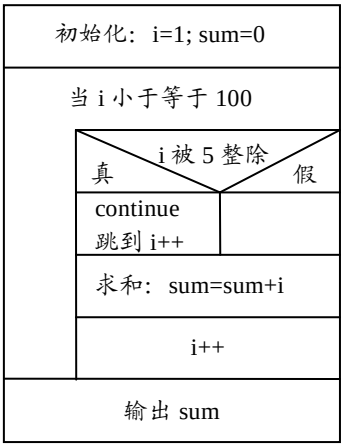


图 3.20　例 3.15 程序 N-S 图

参考代码：

```
#include<stdio.h>
void  main()
{
    int  i, sum=0;
    for(i=1; i<=100; i++)
    {
        if(i % 5 == 0)
            continue;
        sum=sum+i;
```

```
    }
    printf("1 到 100，不能被 5 整除的整数和为%d。\n", sum);
}
```

【说明】① 循环体部分与下列语句等价:

```
{   if(i % 5 == 0)
        ;
    else
      sum=sum+i;
}
```

② continue 语句用在 while 或 do-while 循环结构中，转向下一次循环判断语句，即执行 while 后的表达式，若表达式不包含对循环控制变量的修改，则可能陷入死循环。

【练习 3.22】执行下列程序后 i 的值为_____。

```
#include<stdio.h>
void  main()
{   int  i;
    for(i=0; i<10; i++)
    {  if(i%3==0)
           continue;
       i=i+3;
    }
    printf("%d\n", i);
}
```

【练习 3.23】执行下列程序后得到的情况是_____。

```
#include<stdio.h>
void  main(void)
{   int  i=0;
    while(i<10)
    {
        if(i%3==0)
            continue;
        i+=3;
    }
    printf("%d\n", i);
}
```

3.6 程序控制结构的综合应用

所有的程序都由顺序结构、选择结构和循环结构“搭建”而成。学习 C 语言的目的是通过编制程序解决实际问题，下面通过几个实际应用题，熟练选择结构和循环结构的使用。

【例 3.16】编制判断某一年是否为闰年的程序。

【分析】某一年是闰年的条件是年份数能被 4 整除且不能被 100 整除，或者能被 400 整除。若年份变量为 year，则判断它是否为闰年的逻辑表达式如下：

year%4==0 && year%100!=0 || year%400==0

参考代码：

```
#include<stdio.h>
void main(void)
{
    int year;
    printf("请输入一个年份：");
    scanf("%d", &year);
    if(year%4==0 && year%100!=0 || year%400==0)
        printf("%d 年是闰年。\n", year);
    else
        printf("%d 年不是闰年。\n", year);
}
```

运行实例 1：

```
请输入一个年份：2010
2010 年不是闰年。
```

运行实例 2：

```
请输入一个年份：2012
2012 年是闰年。
```

【思考】怎样采用分支结构的嵌套方式解决该问题。

【例 3.17】判断输入的正整数 n(n>1)是否为素数。

【分析】素数是只能被 1 和它本身整除的 1 以上的正整数。所以可以采用循环的方法，从 2 开始试探是否有数能够整除 n，直到 n−1 为止。让 i 从 2 变化到 n−1，若有一个 i 能够整除 n，就没有继续试探的意义了(采用 break 语句终止循环，此时 i 没有试探结束，i 处在[2，n−1]范围内)。若所有的 i 都不能整除 n，则 n 是素数(此时 i 的值为 n，没有满足循环条件 i<=n−1 就退出了循环)。因此判断 n 是否为素数，关键看退出循环时，i 的值为[2，n−1]内，还是为 n，若 i 的值是前者，则 n 不是素数，否则 n 是素数。

参考代码：

```
#include<stdio.h>
void main()
{
    int i, n;
    printf("请输入一个大于 1 的正整数：");
    scanf("%d", &n);
    for(i=2; i<=n−1; i++)
        if(n % i = = 0)
            break;
    if(i==n )
```

```
            printf("%d 是素数。\n", n);
        else
            printf("%d 不是素数。\n", n);
    }
```

运行实例 1：

请输入一个大于 1 的正整数：13
13 是素数。

运行实例 2：

请输入一个大于 1 的正整数：9
9 不是素数。

【小知识】素数又称质数。一个大于 1 的自然数，如果除了 1 和自身外，不能被其他自然数整除，则这个数就称为素数。换句话说，只有两个正因数(1 和自己)的大于 1 的自然数即为素数。比 1 大但不是素数的数称为合数。1 和 0 既非素数也非合数。素数在数论中有着很重要的地位。

【思考】① 正常退出 for 循环和通过 break 语句退出 for 循环，循环控制变量的值相同吗？

② 尝试用 while 循环或 do-while 循环结构编制例 3.17 要求编制的程序。

【例 3.18】编制程序解决下述的猴子吃桃问题：猴子第 1 天摘下若干个桃子，当即吃了一半，还不过瘾，又多吃了一个；第二天早上又将剩下的桃子吃掉一半后，又多吃了一个；以后每天早上都吃了前一天剩下的一半零一个。到第 10 天早上再想吃时，见只剩下一个桃子了。求猴子第 1 天共摘了多少桃子。

参考代码：

```
#include<stdio.h>
void  main()
{
    int  day, x1, x2;
    day=9;
    x2=1;
    while(day>0)
    { x1=(x2+1)*2;          // 前 1 天的桃子数是后 1 天桃子数加 1 后的 2 倍
      x2=x1;
      day--;
    }
    printf("total=%d\n", x1);
}
```

运行实例：

total=1534

【例 3.19】我国古代数学家张丘建在《算经》中出了一道“百钱百鸡”题：公鸡 5 元 1 只，母鸡 3 元 1 只，小鸡 1 元 3 只，100 元钱买 100 只鸡，问公鸡、母鸡、小鸡各买了多少只？编制程序解决这个百钱百鸡问题。

【分析 1】设公鸡数为 x，母鸡数为 y，小鸡数为 z，则有方程组：

$$\begin{cases} x + y + z = 100 \\ 5x + 3y + z/3 = 100 \end{cases}$$

3 个未知数 2 个方程，解有多组。采用穷举算法对这类问题编程，就是在所有的可能的

情况中找出满足条件的解。

参考代码：

```
#include<stdio.h>
void  main()
{
    int  cock, hen, chick;
    for(cock=0; cock<=100; cock++)
        for(hen=0; hen<=100; hen++)
            for(chick=0; chick<=100; chick++)
                if(cock+hen+chick==100 && cock*15+hen*9+chick==300)  //避免整除误差
                    printf("cock=%d, hen=%d, chick=%d\n", cock, hen, chick);
}
```

上述程序有三重循环，总循环次数为 1000000 次。

【分析 2】由题意可知，公鸡最多不超过 20 只，母鸡最多不超过 33 只，小鸡数为 100 减去公鸡数和母鸡数的和，按上面叙述设计程序，可以大大减少循环次数，提高程序执行效率。修改后代码如下：

```
#include<stdio.h>
void  main()
{
    int  cock, hen, chick;
    for(cock=0; cock<=20; cock++)
        for(hen=0; hen<=33; hen++)
        {
            chick=100-cock-hen;
            if(cock*15+hen*9+chick==300)
                printf("cock=%d,hen=%d,chick=%d\n", cock, hen, chick);
        }
}
```

运行结果如图 3.21 所示。

```
cock=0,hen=25,chick=75
cock=4,hen=18,chick=78
cock=8,hen=11,chick=81
cock=12,hen=4,chick=84
Press any key to continue
```

图 3.21　例 3.19 程序运行结果

【例 3.20】编制程序解决下述问题：从键盘输入一行字符，分别统计出其中英文字母、空格、数字和其他字符的个数 。

参考代码：

```
#include<stdio.h>
void  main()
{
    char  c;
    int  letters=0, space=0, digit=0, other=0;
    printf("请输入一行字符：\n");
    c=getchar()
    while(c != '\n')                //输入回车结束字符串输入
```

```
    {
        if((c>='a' && c<='z') || (c>='A' && c<='Z'))
            letters++;
        else  if(c==' ')
            space++;
        else  if(c>='0' && c<='9')
            digit++;
        else
            other++;
        c=getchar()
    }
    printf("字母数:%d\n 空格数:%d\n 数字数:%d\n 其他字符数:%d\n", letters, space, digit,
        other);
}
```

运行实例：

```
请输入一行字符：
My  room  number  is:502!
字母数：14
空格数：3
数字数：3
其他字符数：2
```

【例 3.21】编制程序解决以下问题：从键盘输入多名学生成绩，直到输入的成绩为负数结束。统计所有成绩的平均分和及格人数。

参考代码：

```
#include<stdio.h>
void  main()
{
    int  grade, cnt1=0, cnt2=0;
    double  avg=0.0;
    printf("请输入多名学生成绩，输入负数退出。\n");
    scanf("%d", &grade);
    while(grade>=0)
    {   if(grade>=60)
            cnt1++;                 //统计及格人数
        cnt2++;                     //统计学生人数
        avg=avg+grade;
        scanf("%d", &grade);
    }
    avg=avg/cnt2;                   //求平均分
    printf("及格人数%d.\n 平均分%.2f.\n", cnt1, avg);
}
```

运行实例：

```
请输入多名学生成绩，输入负数退出。
50 60 70 80 -1
及格人数 3.
平均分 65.00.
```

【思考】如何求　高成绩并输出？如何输出低于平均值的成绩？

习　题　3

1. 有以下程序：

```
void main()
{   int i;
    for(i=0; i<3; i++)
        switch(i)
        {   case 0: printf("%d", i);
            case 2: printf("%d", i);
            default: printf("%d", i);
        }
}
```

运行程序后的输出结果是____。

A. 022111　　B. 021021　　C. 000122　　D. 012

2. 有以下程序：

```
void main()
{
    int a=3, b=4, c=5, d=2;
    if(a>b)
        if(b>c)
            printf("%d", d++ +1);
        else
            printf("%d", ++d +1);
    printf("%d", d);
}
```

运行程序后的输出结果是____。

A. 2　　B. 3　　C. 43　　D. 44

3. 若变量已正确定义，要求程序段完成求5！的计算，不能完成此操作的程序段是____。

A. for(i=1, p=1; i<=5; i++) p*=i;　　B. for(i=1; i<=5; i++)　{p=1;p*=i;}

C. i=1; p=1; while(i<=5)　{p*=i;i++;}　　D. i=1; p=1; do{p*=i;i++;}while(i<=5);

4. 有以下程序

```
void main()
{   int i, s=0;
```

```
    for(i=1; i<10; i+=2)   s+=i+1;
    printf("%d", s);
}
```

运行程序后的输出结果是____。

A. 自然数1～9的累加和　　B. 自然数1～10的累加和

C. 自然数1～9中奇数之和　　D. 自然数1～10中偶数之和

5. 以下程序段中与语句“k=a>b?(b>c?1:0):0;”功能等价的是_____。

A.
```
if((a>b)&&(b>c)) k=1;
else k=0;
```

B.
```
if((a>b)||(b>c)) k=1;
else k=0;
```

C.
```
if(a<=b) k=0;
else if(b<=c) k=1;
```

D.
```
if(a>b) k=1;
else if(b>c) k=1;
else k=0;
```

6. 有以下程序：

```
void main( )
{  int i;
   for(i=1; i<6; i++)
   {   if(i%2)   {printf("#");   continue;}
       printf("*");
   }
   printf("\n");
}
```

运行程序后的输出结果是____。

A. #*#*#　　B. #####　　C. *****　　D. *#*#*

7. 有以下程序：

```
void main()
{  int k=4,n=0;
   for( ; n<k;)
   {  n++;
      if(n%3!=0)    continue;
      k--;}
   printf("%d,%d", k, n);
}
```

运行程序后的输出结果是____。

A. 1, 1　　B. 2, 2　　C. 3, 3　　D. 4, 4

8. 有如下程序：

```
void main()
{  int i, sum;
   for(i=1; i<=3; sum++)   sum+=i;
   printf("%d\n", sum);
}
```

运行程序后的结果是____。

A. 输出6　　B. 输出3　　C. 输出0　　D. 陷入死循环

9. 有以下程序：

```
#include<stdio.h>
void  main()
{   int  x=1, y=2, z=3;
    if(x>y)
        if(y<z)    printf("%d", ++z);
        else       printf("%d", ++y);
    printf("%d\n", x++);
}
```

运行程序后的输出结果是____。

A. 331　　B. 41　　C. 2　　D. 1

10. 有以下程序段：

```
int  a=3, b=5, c=7;
if(a>b)     a=b;     c=a;
if(c!=a)    c=b;
printf("%d, %d, %d", a, b, c);
```

运行程序后的结果是____。

A. 程序段有语法错误　　B. 输出3, 5, 3　　C. 输出3, 5, 5　　D. 输出3, 5, 7

编制程序，完成 11～18 题的要求。

11. 从键盘上输入一个英文字母，如果输入的英文字母为 y 或者 Y，则输出 yes；如果输入的英文字母为 n 或者 N，则输出 No。

12. 输出 100～999 之间的水仙花数。(三位水仙花数是指一个三位数，各位的立方和等于该三位数，如 153，370 等。)

13. 从键盘输入一个正整数，如果该数为素数，则输出该素数，否则输出该数的所有因子(除去 1 与自身)。

14. 显示输出能写成两个数平方和的所有三位数。

15. 从键盘输入 20 个实数，分别计算并输出所有正数之和，所有负数之和，所有数的绝对值之和，正数的个数和负数的个数。

16. 求 2 到 500 之间的所有亲密数对。亲密数对的定义为：如果 M 的因子(不含 1 和 M)之和为 N，且 N 的因子(不含 1 和 N)之和为 M，则称 M 与 N 为一对亲密数。

17. 某参观团按以下条件限制从 A、B、C、D 和 E 五个地方中选定若干参观点。

① 如果去 A 地，则必须去 B 地；

② D 和 E 两地中只能去一地；

③ B 和 C 两地中只能去一地；

④ C 和 D 两地要么都去，要么都不去；

⑤ 如果去 E 地，则必须去 A 和 D 地。

问该参观团能去哪些地方？

18. 用对分法求方程 $2x^3-4x^2+6x-1=0$ 在区间[−10，10]上的实根，精度要求 $\varepsilon=10^{-6}$。

第4章　数　　组

前几章使用的数据都是基本类型(整型、实型、字符型)的数据，程序中只涉及少量的变量。在实际问题中往往会有很多数据，只定义几个变量不能解决问题。回顾例3.21，统计成绩平均分和及格人数。如果要求输出低于平均分的成绩，就首先要计算平均分，然后将输入的每个成绩分别和平均分比较，低于平均分则输出该成绩。要将每个成绩都跟平均分进行比较，显然需要在输入的时候将每个成绩都保存下来。这就需要定义多个相同类型的变量。

C语言中除了基本数据类型外，还有构造数据类型，数　就是一种构造数据类型。

数　是具有一定顺序的若干相同类型变量的集合体，　成数　的变量称为该数　的元素。每个数　元素都是一个变量，它们有相同的数据类型，这些变量是有顺序的。每个数　都有一个数　名。

知 识 结 构

1. 一维数组
   ① 一维数组的定义和引用
   ② 一维数组的初始化
   ③ 一维数组编程实例
2. 一维字符数组和字符串
   ① 一维字符数组
   ② 字符串
3. 二维数组
   ① 二维数组的定义和引用
   ② 二维数组的初始化
   ③ 二维数组编程实例

4.1　一 维 数 组

4.1.1　一维数组的定义和引用

1. 一维数组的定义

定义一维数组的一般形式为：

类型说明符　数组名[常量表达式]；

例如：

语句“int a[10];”定义了一个数组 a，它含有 10 个整型元素。

语句“char c[20];”定义了一个数组 c，它含有 20 个字符型元素。

定义一个数组后，系统会在内存中开辟一段连续的空间。数组名就是这段空间的首地址。空间的大小=数组元素个数×数组元素类型占内存的字节数，空间大小的单位是字节。

例如对于上面定义的数组a[10]，系统会在内存中给它分配 10×4=40 个字节的空间(假设系统中一个整型变量数据占 4 个字节)。

【说明】① 数　定义中的类型说明符，指的是数　元素的类型。

② 数　名需　循标识符的命名规则，且不能和其他变量同名。

③ 中括号“[”和“]”中的常量表达式表示数　中元素的个数，即数　的长度或大小，可以是常量或符号常量，但不能是变量。以下程序段是一个典型的错误定义方式。

```
int  n=10;
int  a[n];    //n 是变量，不允许用来定义数　的长度
```

数　的大小不能依赖于程序运行过程中变量的值，即 C 语言不允许对数　的大小　动态定义。

2. 一维数组的引用

C 语言规定只能逐个引用数组元素而不能一次引用整个数组。

一维数组元素引用的一般形式为：

数组名[下标]

引用数组元素时，数组元素本身相当于一个该类型的变量，因此，对数组元素的操作类似于对同类型变量的操作。

例如，下面的程序段可以交换整型数组元素 a[0]和 a[1]的值：

```
int  t;
t=a[0];
a[0]=a[1];
a[1]=t;
```

下面代码用于从键盘输入一个整数存放在整型数组元素 a[i]中。

```
scanf("%d", &a[i]);
```

【注意】① 数　元素和变量一样必须先定义后使用，即数　名必须是已经定义过的。

② 下标可以是整型常量或整型表达式，其取值范围从 0 开始，到“数　元素个数-1”，引用时下标不能越界。例如下述程序段有错误:

```
int  a[10];
printf("%d", a[10]);          //错误，下标越界
```

因为数　a 可以引用的元素是 a[0]，a[1]，a[2]，…，a[9]，没有 a[10]这个元素。

3. 一维数组的存储顺序

数组在内存中占一段连续内存单元，数组元素在内存中顺次存放，它们的地址是连续的。

例如：具有 10 个元素的整型数组 a 在内存中的存放次序如图 4.1 所示。

0	1	2	3	4	5	6	7	8	9
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]

图 4.1　一维数组存放顺序

图 4.1 表示 a[0]，a[1]，…，a[9]在内存中顺序存放，且其值分别是 0，1，2，3，4，5，6，7，8，9。

【注意】① 数 名是数 首元素的地址，即使用 “scanf("%d", a);”语句也可以完成对 a[0]的输入，但通常不这样使用。

② 数 名是一个常量，不能被赋值。

例如 “a=0;” 或 “a={1, 2, 3, 4};”都是错误的语句。

4.1.2 一维数组的初始化

在定义数组时可以对数组元素赋初值，所赋初值放在赋值号后的一对花括号中(不可为空)，初值之间用逗号分隔，编译系统按这些值的顺序从第一个元素起依次赋值，数值类型必须与说明类型一致。

对数组初始化的一般形式为：

类型名 数组名[数组长度] = {初值表};

下面举例说明如何给数组元素赋初值。

① 在定义数组时对数组全部元素赋初值。例如：

```
int a[10]={0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

② 只给一部分元素赋初值。当初值个数少于定义的数组元素个数时，系统将自动给后面的数值型元素赋 0 值，字符型元素赋 '\0'(ASCII 码为零的字符)值。例如：

```
int a[10]={0, 1, 2, 3, 4};
```

相当于 a[0]=0，a[1]=1，a[2]=2，a[3]=3，a[4]=4，其他元素 a[5]～a[9]的值全为 0。

③ 在对全部数组元素赋初值时，可以不指定数组长度。例如：

```
int a[]={1, 2, 3, 4,5 };
```

等价于

```
int a[5]={1, 2, 3, 4, 5};
```

表示定义了一个具有 5 个整型元素的数组 a，同时给每个元素都赋了初值。

④ 数组和变量一样都有静态和动态之分，对 static(静态)数组不进行初始化时，如果数组元素是整型、实型等数值类型数据，系统自动赋初值为 0；如果数组元素是字符型数据，系统自动赋初值为空字符'\0'；定义数组时不加 static，则默认是 auto 类型，对 auto 类型数组如果不进行初始化时，系统自动为其赋随机值。

4.1.3 一维数组编程实例

一维数组编程中最常用的结构是 for 循环，因为数组下标从 0 至 “数组长度-1”，所以常用循环控制变量做下标，用 for 循环来处理数组。例如，输入一个正整数 n(1<n≤10)，再输入 n 个数存到数组中，代码如下：

```
int a[10];                  //定义 1 个数组 a，它有 10 个整型元素
printf("Enter  n: ");
scanf("%d", &n);
printf("Enter %d  integers: ", n);
for(i = 0; i < n; i++)                    // 下标 i 从 0 变化到 n-1
    scanf("%d", &a[i]);
```

【例 4.1】输入 10 个学生某门课程的成绩，显示 10 个学生平均分和低于平均分的成绩。

【分析】可以通过输入一个数就累加一个数的办法来求学生的总分，进而求出平均分。要显示低于平均分的成绩，必须把 10 个学生的成绩都保留下来，采用数组存放，然后逐个和平均分比较。

参考代码：

```
#include<stdio.h>
#define  N  10
int  main(void)
{
    double  a[N], total=0, average;     int  i;
    /*  统计成绩总和  */
    for(i=0; i<N; i++)
    {
        printf("Enter  %d  grade:", i);
        scanf("%lf", &a[i]);
        total=total+a[i];
    }
    /*  求平均成绩*/
    average=total/N;
    printf("Grade  average  is  %.2f\n",average);
    /*  输出低于平均分成绩  */
    for(i=0; i<N; i++)
        if(a[i]<average)
            printf("a[%d]=%.2f<%.2f\n", i, a[i], average);

    return 0;
}
```

运行实例：

```
Enter 0 grades:80
Enter 1 grades:70
Enter 2 grades:60.5
Enter 3 grades:50
Enter 4 grades:80
Enter 5 grades:90
Enter 6 grades:75
Enter 7 grades:85
Enter 8 grades:92
Enter 9 grades:95
Grade average is 77.75
a[1]=70.00<77.75
```

a[2]=60.50<77.75
a[3]=50.00<77.75
a[6]=75.00<77.75

【思考】计算学生平均成绩可不可以不用数 ？查找低于平均分的成绩可不可以不用数？

【例 4.2】Fibonacci 数列满足以下关系：f(1)=1，f(2)=1，当 n>2 时，f(n)=f(n−2)+f(n−1)。输出 Fibonacci 数列前 20 项，每 10 个数一行。

【分析】可以定义数组来存放数列中前 20 个数，数列中前两个数已知，可以在数组定义时给前两个元素赋初值，其他元素由前两个元素逐步递推得到，最后输出数列。

参考代码：

```
#include<stdio.h>
int  main()
{
    int  i;
    int  f[20]={1, 1};              //初始化第 0、1 个数
    for(i=2; i<20; i++)
        f[i]=f[i−2]+f[i−1];         //求 f[2]～f[19]
    for(i=0; i<20; i++)             //输出数组元素，每行 10 个数
    {
        printf("%6d", f[i]);        //设置输出宽度为 6
        if((i+1)%10==0)
            printf("\n");
    }
    return  0;
}
```

运行实例：

```
     1      1      2      3      5      8     13     21     34     55
    89    144    233    377    610    987   1597   2584   4181   6765
```

【思考】例 4.2 不用数 如何实现，有什么区别？

【例 4.3】从键盘输入 10 个数，输出最小值和它的下标，将该最小值和第一个数交换，输出交换后的结果。

【分析】利用数组存放 10 个数，数组定义为“int a[10]”，用变量 index 记录最小值对应的下标，则最小值就是 a[index]。

参考代码：

```
#include<stdio.h>
#define  N  10                  //定义符号常量 N
int  main()
{
    int a[N], i, index, temp;
```

```
    //输入 N 个数，并存到数组 a 中
    printf("please input %d numbers\n", N);
    for(i=0; i<N; i++)
        scanf("%d", &a[i]);

    index=0;        //index 用来存放最小值的下标，初值是第一个元素下标 0

    //在 a[0]～a[N-1]中找最小值的下标放到 index 中
    for(i=1; i<N; i++)
        if(a[i]<a[index])
            index=i;

    // 将找到的最小值与数组第 1 个数 a[0]交换
    temp=a[index];
    a[index]=a[0];
    a[0]=temp;

    //输出交换后的结果
    printf("the array after change:\n");
    for(i=0; i<N; i++)
        printf("%3d", a[i]);
    printf("\n");

    return 0;
}
```

运行实例：

```
please input 10 numbers
4 5 3 8 9 1 0 2 6 7
the array after change:
  0  5  3  8  9  1  4  2  6  7
```

【练习 4.1】输入一个正整数 n(1<n≤10)，再输入 n 个整数存入一维数组，先输出最大值及其下标(设最大值唯一)，再将最大值与最后一个数交换，并输出交换后的 n 个数。

【例 4.4】从键盘输入 10 个数，对 10 个数从小到大排序，并输出排序后的数。

【分析】例 4.3 从 10 个数(a[0]～a[9])中找最小值并与第 1 个数(即 a[0])交换，得到 10 个数中的第 1 个数最小的排列；如果要对 10 个数排序，则可以利用类似的方法对剩余数据接着进行处理，假设把在 10 个数中找最小值并与第一个数交换的操作称为第 0 次，那后面的处理顺序如下。

第 1 次：对结果中除第 1 个数外剩下的 9 个数(a[1]～a[9])利用类似的方法找最小值，然后与剩余的 9 个数中第 1 个数(即 a[1])交换，得到 10 个数中前两个数有序的排列；

第 2 次：对结果中除前两个有序数外剩下的 8 个数(a[2]～a[9])利用类似的方法找最小值，然后与剩余的 8 个数中第 1 个数(即 a[2])交换，得到 10 个数中前 3 个数有序的排列；

……

第 i 次：对结果中除前 i 个有序数外剩下的 10−i 个数(a[i]～a[9])利用类似的方法找最小值，然后与剩余的 10−i 个数中第 1 个数(即 a[i])交换，得到的 10 个数中前 i+1 个数有序的

排列；

……

第 8 次，对结果中除前 8 个有序数外剩下的 2 个数(a[8]～a[9])利用类似的方法找最小值，然后与剩余的 2 个数中第 1 个数(即 a[8])交换，得到的 10 个数中前 9 个数有序的排列，最后一个便是最大值，这样就得到 10 个有序排列的数。

上面的排序算法称为选择法排序。例如，对“3、5、2、8、1”五个数排序，上面的排序过程如图 4.2 所示。

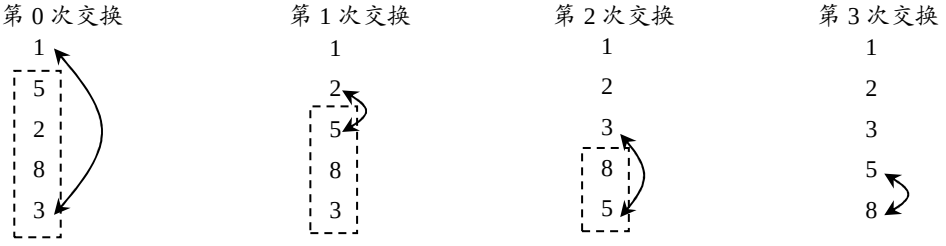


图 4.2　选择法排序过程

选择法排序的思路是：每次从剩余的无序数中找出最小(大)的数，然后将该最小(大)数与无序数中的第一个数交换。选择法排序的 N-S 图如图 4.3 所示。

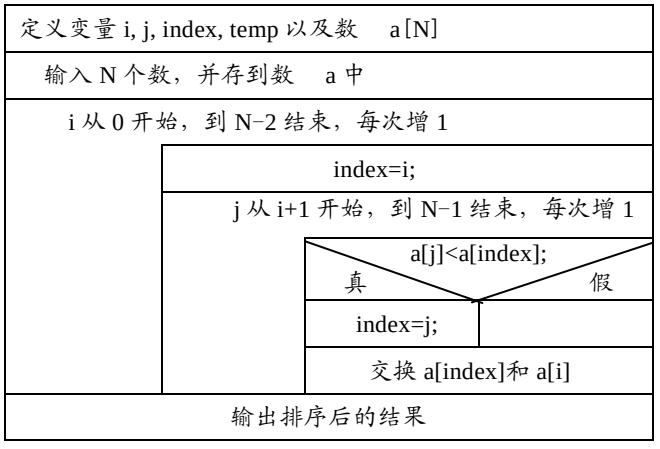


图 4.3　选择法排序 N-S 图

参考代码：

```
#include<stdio.h>
#define  N  10                    //定义符号常量 N，对 N 个数排序
int main()
{
    int  a[N], i, j, index, temp;

    //输入 N 个数，并存到数组 a 中
    printf("Please  input  %d  numbers：\n", N);
    for(i=0; i<N; i++)
        scanf("%d", &a[i]);
```

```
    //对数组 a 中的数排序，每循环一次选出数组第 i 到最后数中最小的数，i 从 0 到
    //N-2 取值
    for(i=0; i<N-1; i++)
    {
    //index 用来存放最小值的下标，初值是未排序的数组元素
    // a[i]～a[N-1]中第一个元素的下标 i
        index=i;
    //在 a[i]～a[N-1]中找最小值的下标放到 index 中
        for(j=i+1; j<N; j++)
            if(a[j]<a[index])
                index=j;
    //将找到的最小值与未排序的序列中第一个数 a[i]交换
        temp=a[index];
        a[index]=a[i];
        a[i]=temp;
    }
    //输出排序后的结果
    printf("The array after sort:\n");
    for(i=0; i<N; i++)
        printf("%3d", a[i]);
    printf("\n");
    return 0;
}
```

运行实例：

```
Please input 10 numbers：
4 5 3 8 9 1 0 2 6 7
The array after sort:
  0  1  2  3  4  5  6  7  8  9
```

【例 4.5】从键盘输入一个正整数 n(1<n≤10)，再输入 n 个数，用冒泡法从小到大排序后输出。

【分析】冒泡法排序的思路是：将相邻两个数比较，将小的调到前面，大的沉到后面。假设程序中输入 n 的值为 6，输入的 6 个数分别为 9，8，5，4，6，0，则用冒泡法进行排序处理过程如下。

第 1 趟：对数组中 6 个数两两进行比较，首先比较 a[0]和 a[1]，由于它们不符合次序要求，因此进行交换；然后比较 a[1]和 a[2]，由于它们不符合次序要求，也要进行交换；再比较 a[2]和 a[3]，以此类推……若相邻元素符合次序要求，不进行交换；若不符合次序要求，则要对它们进行交换。第 1 趟完成后，数组中最大的数 9 就沉到最后面，存放到 a[5]中。

第 2 趟：对数组中前 5 个数两两进行比较，首先把 a[0]和 a[1]进行比较，并按第 1 趟那样进行类似处理，以此类推。第 2 趟完成后，数组中第二大的数 8 就沉到前 5 个数的最后面，

存放到 a[4]中。

……

第 5 趟：对数组中前 2 个数进行两两比较，若不符合次序要求，进行交换。至此，数组中所有元素已按从小到大排好序。

上面过程每一趟的排序结果如图 4.4 所示。

第 1 趟						第 2 趟	第 3 趟	第 4 趟	第 5 趟
9	8	8	8	8	8	5	4	4	0
8	9	5	5	5	5	4	5	0	4
5	5	9	4	4	4	6	0	5	5
4	4	4	9	6	6	0	6	6	6
6	6	6	6	9	0	8	8	8	8
0	0	0	0	0	9	9	9	9	9

图 4.4　冒泡排序过程

冒泡法排序的 N-S 图如图 4.5 所示。

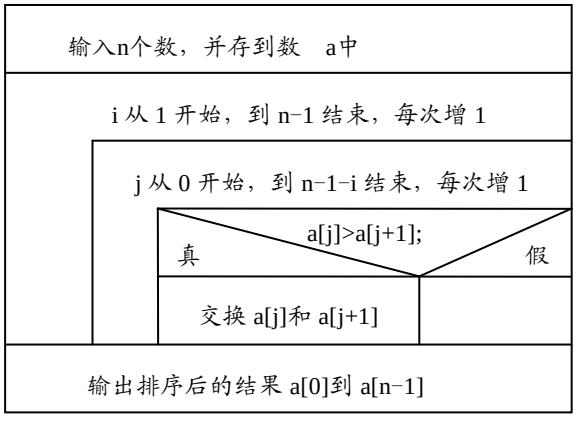


图 4.5　冒泡法排序 N-S 图

参考代码：

```
#include<stdio.h>
int  main()
{
    int  a[10];
    int  i, j, t, n;

    printf("Input  n: ");
    scanf("%d", &n);
    //从键盘输入 n 个整数
    printf("Input  %d  numbers: ", n);
    for(i=0; i<n; i++)
        scanf("%d", &a[i]);

    //用冒泡法开始排序
```

```
        for(i=1; i<n; i++)
            for(j=0; j<n-i; j++)
                if(a[j]>a[j+1])
                {
                    t=a[j];
                    a[j]=a[j+1];
                    a[j+1]=t;
                }
        //输出排序后的结果
        printf("The sorted numbers:");
        for(i=0; i<n; i++)
            printf("%d ", a[i]);
        printf("\n");

        return 0;
    }
```

运行实例：

```
Input n: 8
Input 8 numbers: 9 8 5 4 6 0 7 2
The sorted numbers: 0 2 4 5 6 7 8 9
```

【练习4.2】若要定义一个具有5个元素的整型数组，以下定义语句中错误的是_____。

A. int a[5]={0};　　B. int b[]={0, 0, 0, 0, 0};

C. int c[2+3];　　D. int i=5,d[i];

【练习4.3】若有定义语句“int a[]={5, 4, 3, 2, 1}, i=4;”，则下面对a数组元素的引用中错误的是_____。

A. a[--i]　　B. a[2*2]　　C. a[a[0]]　　D. a[a[i]]

【练习4.4】有以下程序：

```
#include<stdio.h>
void main()
{
    int a[5]={1, 2, 3, 4, 5}, b[5]={0,2,1,3,0},i,s=0;
    for(i=1; i<5; i++)    s=s+a[b[i]];
    printf("%d\n", s);
}
```

运行程序后的输出结果是_____。

A. 6　　B. 10　　C. 11　　D. 15

【练习4.5】有以下程序：

```
#include<stdio.h>
int main()
{
```

```
    int  s[12]={1, 2, 3, 4, 4, 3, 2, 1, 1, 1, 2, 3}, c[5]={0}, i;
    for(i=0; i<12; i++)    c[s[i]]++;
    for(i=1; i<5; i++)     printf("%d", c[i]);
    printf("\n");
    return  0;
}
```

运行程序后的输出结果是_____。

A. 4332　　B. 4321　　C. 1234　　D. 2334

【练习 4.6】编程解决：输入一个正整数 n(1<n≤10)，再输入 n 个浮点数存入一维数组，然后输出该数组中的最小值及其下标。

4.2　一维字符数组和字符串

C 语言中，单个字符常量用单引号括起来，单个字符变量用 char 类型定义，那么包含多个字符的常量应该怎样表示？存放多个字符的变量应该用什么类型定义？C 语言中利用一维字符数组来存放多个字符。

4.2.1　一维字符数组的定义和初始化

一维字符数组用于存放多个字符型数据，它的定义、引用和初始化与其他类型的一维数组一样。

1. 一维字符数组的定义

一维字符数组定义的一般形式为：

```
char  数组名[常量表达式];
```

例如“char str[20];”表示定义了一个字符型数组，数组名是 str，含有 20 个字符型元素。

2. 一维字符数组的引用

一维字符数组元素引用的一般形式为：

```
数组名[下标]
```

其中下标的取值范围是从 0 到数组长度−1。如上面定义的数组 str 的下标的取值范围是 0～19，可以引用的元素是 str[0]，str[1]，…，str[19]。

3. 一维字符数组的初始化

一维字符数组与整型数组一样，也可以在定义的同时进行初始化。例如：

```
char t[5]={'H', 'a', 'p', 'p', 'y'};
```

初始化后，相当于对数组 t 中各元素进行了如下的赋值：t[0]= 'H'，t[1]='a'，t[2]='p'，t[3]='p'，t[4]='y'。又如：

```
static  char  s[6]={'H', 'a', 'p', 'p', 'y'};
```

等价于

```
char  s[6]={'H', 'a', 'p', 'p', 'y'};
```

对数组中部分元素初始化时，对未初始化的元素系统自动赋 0 值，所以上述初始化语句等价于：

```
static char s[6]={'H', 'a', 'p', 'p', 'y', 0};
```

或

```
char s[6]={'H', 'a', 'p', 'p', 'y', 0};
```

整数 0 代表字符'\0'，即 ASCII 码为 0 的字符，表示空字符或空操作。所以，上述初始化语句还等价于：

```
static char s[6]={'H', 'a', 'p', 'p', 'y', '\0'};    或    char s[6]={'H', 'a', 'p', 'p', 'y', '\0'};
```

与其他一维数组初始化相同，如果对全部元素都赋了初值，可以省略数组长度。例如，上述语句等价于：

```
static char s[]={'H', 'a', 'p', 'p', 'y', '\0'};
```

该数组长度为 6，相应内存单元的存储内容如图 4.6 所示：

H	a	p	p	y	\0
s[0]	s[1]	s[2]	s[3]	s[4]	s[5]

图 4.，6　字符数组的存储

4.2.2　字符串概念

字符串即一串字符，程序中用来处理多个字符的问题时，就要用到字符串。

字符常量是用一对单引号括起来的单个字符，字符串常量是用一对双引号括起来的字符序列，也就是一串字符。每一个字符串常量都有一个结束标志'\0'。例如，字符串"Happy"由六个字符组成，分别是'H'，'a'，'p'，'p'，'y'和'\0'。其中前五个字符是字符串的有效字符，'\0'是字符串结束符。

字符串的长度就是有效字符的个数，例如，"Happy"的有效长度是 5。

C 语言使用一维字符数组来存储字符串，由于字符串有结束符'\0'，它的存储和处理又有自身的特点，所以 C 语言中将字符串看作一个特殊的一维字符数组来处理。学习字符串的存储和处理一定要明确两点：

① 字符串存储在一维字符数组中。

② 字符串由有效字符和字符串结束符'\0'组成。

4.2.3　字符串存储

将字符串存储到一维字符数组中，主要有三种方式。

1. 字符数组初始化

对字符数组初始化时，可以逐个元素赋初值，也可以用字符串常量对字符数组初始化。

① 逐个元素赋初值，例如

```
char s[6] = {'H','a','p','p','y'};
```

等价于

```
char s[6] = {'H','a','p','p','y','\0' };
```

数组 s 中存放了字符串"Happy"。

② 用字符串常量对字符数组初始化，例如

```
char s[6] = {"Happy"};
```

等价于

```
char s[6] = "Happy";          或          char s[] = "Happy";
```

用字符串常量给字符数组初始化时，系统自动加上空字符'\0'（结束符）；所以用字符串

常量初始化字符数组时必须满足：数组长度大于或等于字符串的有效长度+1。例如，字符串"Happy"的有效长度是 5，存储它的数组的长度至少应该是 6。如果上面的数组 s 定义为

```
char s[5]= "Happy";
```

那么由于数组 s 定义长度太小，字符串的结束符未存储，则数组 s 中存放的不是字符串，只是'H'、'a'、'p'、'p'、'y'五个字符。

如果数组长度大于字符串的有效长度+1，则数组中除了存储的字符串，还有其他内容，即字符串只占用了数组的一部分，例如：

```
char s[20]= "Happy";
```

只对数组的前六个元素赋初值，其他元素的值都为'\0'。但是处理字符串时不会考虑其他元素的值。由于字符串遇到'\0'结束，所以，数组中第一个'\0'前的所有字符和第一个'\0'构成了字符串"Happy"，也就是说，第一个'\0'之后的其他数组元素与该字符串无关。

同样采用按元素初始化的方式，若数组长度大于赋值的字符个数，例如：

```
char s[6] = {'H', 'a', 'p', 'p', 'y'};
```

等价于

```
char s[6] = {'H', 'a',' p', 'p', 'y', '\0'};
```

其他未赋初值的元素都为'\0'。实际操作字符串时，不会考虑所有的元素，只考虑数组中第一个'\0'和第一个'\0'前的所有字符。

2. 赋值

将字符串存入数组，除了上面介绍的数组初始化方法外，还可以采用赋值的方法。字符串的赋值和普通变量不同，不能一次给整个字符数组赋值，只能给单个元素赋值。例如用语句“char str[80];”定义了字符数组后，下面是几种常见的错误的赋值方法：

```
str[80]= "Happy";       //错误，定义数组后，引用时下标只能从 0 到 79
str[]="Happy";          //错误，str[]不代表任何意义
str="Happy";            //错误，str 是数组名，将字符串常量赋值给数组名，无意义
```

正确的赋值语句如下：

```
str[0]='H';  str[1]='a';  str[2]='p';  str[3]='p';  str[4]='y';  str[5]='\0';
```

【注意】① 因为字符串有结束符'\0'，要使字符数 str 中存放的是一个字符串，必须给字符数 赋值结束符'\0'，这样字符数 中存放的才是字符串。

例如对于包含 6 个元素的字符数 s[6]，程序段“char s[6]; s[0]='a'; s[1]='\0';”与语句“char s[6]= "a";” 用相同，都是将字符串"a"存入数 s 中。

② 注意区分"a"和'a'。前者是字符串常量，代表两个字符'a'和'\0'，用一维字符数 存放；后者是字符常量，只有一个字符，用字符变量存放。

3. 键盘输入

还可以通过从键盘输入的方式，将字符串存入一维字符数组中，可以通过循环，逐个输入字符，也可以一次输入整个字符串。

① 逐个输入字符元素：

利用格式化输入函数 scanf()或字符输入函数 getchar()，可以逐个字符输入数组中各元素。要注意字符串结束符'\0'代表空操作(空字符)，无法输入，所以输入时，应先设定一个输入结束符，在程序中将输入结束符转换为字符串结束符'\0'。例如，“输入一个以回车结束的字符串”的代码如下：

```
char str[80];
```

```
int i=0;
while((str[i] = getchar()) != '\n')
    i++;
str[i] = '\0';                    // 将输入的结束符转换为字符串结束符
```

【注意】要将　后的输入结束符'\n'，转换为字符串结束符'\0'。

② 整个字符串输入：

方法 1：利用 scanf()函数，采用%s 格式符，输入时，碰到空格或回车结束。例如：

```
char  str[80];
scanf("%s", str);
```

【注意】数　名表示数　的首地址，所以输入时，数　名 str 前面不必加&。

若一次输入多个字符串，如何区分输入的数据保存到哪个字符数组中？例如下面的代码：

```
char  str1[5], str2[5], str3[5];
scanf("%s%s%s", str1, str2, str3);
```

如果输入数据为“how　are　you？”则在内存中的存放如图 4.7 所示。

str1：	h	o	w	\0
str2：	a	r	e	\0
str3：	y	o	u	\0

图 4.7　**str1**、**str2**、**str3** 的存储

【注意】用%s 输入时，遇到空格或回车便认为一个字符串输入结束。

方法 2：使用字符串输入函数 gets(字符数组名)。

字符串输入函数的作用是从键盘输入一个字符串(以回车结束)，放到数组中。例如：

```
char  str[80];
gets(str);
```

4.2.4　字符串输出

1. 逐个字符输出

利用格式化输出函数 printf()或字符输出函数 putchar()，可以逐个字符输出数组中各元素。例如，若 s 数组定义为“char s[6]= "Happy";”输出字符串的实现代码为：

```
for(i=0; s[i]!= '\0'; i++)
    printf("%c", s[i]);
```

或

```
for(i=0; s[i]!= '\0'; i++)
    putchar(s[i]);
```

2. 整个字符串输出

① 使用 printf()函数是采用%s 格式符，可以一次输出整个字符串。输出时，碰到字符串结束符便结束，例如语句“printf("%s", s);”的作用是输出 s 中存放的字符串。

【注意】用格式符%s 输出，无论数　有多少个元素，只要遇到'\0'便结束。

② 使用字符串输出函数：

字符串输出函数的格式为

puts(字符数组名/字符串常量)

作用是输出字符串(以'\0'结束的字符序列)，输出完成后自动换行。例如：

puts("ok");　　等价于　　printf("ok\n");

【说明】字符串输入函数 gets()和字符串输出函数 puts()定义在 stdio.h 文件中，所以使

用字符串输入输出函数时，必须添加编译预处理命令“#include<stdio.h>”。另外，使用字符串处理函数必须保证处理的字符数 中存储的是字符串，即必须有字符串结束标志'\0'。

4.2.5 字符串的处理

由于字符串存储在一维字符数组中，因此，处理字符串实际上就是处理一维字符数组，但是处理字符串和处理普通的字符数组不同，处理普通的字符数组，数组元素的个数是确定的，一般用下标控制循环；而处理字符串是处理有效字符，由于没有显式地给出有效字符的个数，只规定在字符串结束符'\0'之前的字符是字符串的有效字符，所以一般用判断结束符'\0'来控制循环。

若有存放字符串的数组定义“char str[80];”则处理字符串时，循环结束条件由普通数组的下标控制 i<80，改为 str[i]!= '\0'。

【例 4.6】输入一个以回车结束的字符串(少于 80 个字符)，统计输出其中大写字母的个数，并将输入的字符串原样输出。

【分析】字符串的输入在上面已经介绍，重点注意逐个输入字符时，由于'\0'是空操作，无法输入，所以用回车作为字符串的结束符，最后把结束符'\n'转化为字符串结束符'\0'。要统计大写字母的个数，则要对字符串中的有效字符逐个进行判断，是大写字母则计数器增 1，重点注意判断字符串有效字符，而不是数组的所有元素。

参考代码：

```
#include<stdio.h>
int  main(void)
{   int  count, i;
    char  str[80];
    printf("Enter  a  string: ");
//输入字符串
     i = 0;
     while((str[i] = getchar()) != '\n')
          i++;
     str[i] = '\0';                          // 将输入结束符转换为字符串结束符
    count = 0;
    for(i = 0; str[i] != '\0'; i++)          // 用判断是否为字符串结束符来控制循环
         if(str[i] >= 'A'&& str[i] <= 'Z' )
              count++;
    printf("count = %d\n", count);
    //输出字符串
    for(i = 0; str[i] != '\0'; i++)
        putchar(str[i]);
    return  0;
}
```

运行实例：

```
Enter a string: A12Bc45
count = 2
A12Bc45
```

【思考】可不可以把 for 循环的循环条件“str[i] != '\0'”改为“i<80”？如果改了，会出现什么结果？

【说明】① 处理字符串时，不管字符数　的长度为多少，只考虑字符串结束符'\0'和结束符之前的有效字符。所以处理字符串时，常用“str[i] != '\0'”　为循环条件。若改为用下标变量控制，对结束符后的循环没有意义，甚至影响程序执行结果。例如例 4.6 的“输出字符串”模块中，如果把 for 循环的结束条件“str[i] != '\0'”改为“i<80”，则数　str 的 80 个元素都会输出，输出字符串后，结束符后的元素以随机值形式输出。

② 若已知字符串的有效长度 length，则处理字符串的循环条件可以改为 i<length，C 语言提供了求字符串有效长度的 strlen()函数，若 str 数　定义为“char str[6]= "Happy";”则可以通过调用函数 strlen(str)，求出字符串"Happy"的有效长度是 5。strlen()函数定义包含在头文件 string.h 中，所以使用该函数时必须加上编译预处理命令#include<string.h>。

【例 4.7】输入一行字符(少于 80 个)，统计其中有多少个单词。单词之间用空格分隔，且空格可以连续出现。

【分析】单词的个数由空格出现的次数决定(连续出现多个空格作为出现一次，一行开头的空格不统计在内)。如果某一个字符为非空格字符，而它前面的字符是空格，则表示新单词开始了，此时单词数累加 1。如果当前字符不是空格，而它前面的字符也不是空格，则意味着仍然是原来单词的继续，单词数不应累加 1。

参考代码：

```
#include<stdio.h>
int main()
{
    char str[80];
    int i, count=0, word=0;
    printf("请输入一行字符：");
    gets(str);
    //统计单词个数
    for(i=0; str[i]!='\0'; i++)
    {
        if(str[i]==' ')      word=0;
        else if(word==0)
        {
            count++;
            word=1;
        }
    }
    printf("单词个数是：%d \n", count);
    return 0;
}
```

运行实例：

请输入一行字符：I am a boy

单词个数是：4

【说明】程序中用变量 count 统计单词个数，用变量 word 为判断是否为单词的标志。当前字符前面一个字符是否空格可以从 word 的值看出来，若 word 等于 0，则表示前一个字符是空格；如果 word 等于 1，则表示前一个字符不是空格。

【例 4.8】输入一个以回车结束的字符串(少于 10 个字符)，字符串由数字字符组成，将该字符串转换成整数后输出。

【分析】用一维字符数组实现字符串的存储和运算，数组长度取上限 10，以'\n'作为输入结束符。若将字符串"123"存入数组 s，则字符串转换成整数 n 的循环执行过程如图 4.8 所示。

i	s[i]	s[i]-'0'	n=n*10+s[i]-'0'
0	'1'	1	n=0*10+1=1
1	'2'	2	n=1*10+2=12
2	'3'	3	n=12*10+3=123
3	'\0'		

图 4.8 字符串转换成整数的过程

参考代码：

```
#include<stdio.h>
int main(void)
{
    int i, n;
    char s[10];
    // 输入字符串
    printf("Input a string: ");          // 输入提示
    i = 0;
    while((s[i] = getchar()) != '\n')
        i++;
    s[i] = '\0';
    // 将字符串转换为整数
    n = 0;
    for(i = 0; s[i] != '\0'; i++)
        if(s[i] <= '9' && s[i] >= '0')
            n = n * 10 + (s[i] - '0');
        else                             // 遇非数字字符结束转换
            break;
    printf("Digit = %d\n", n);
    return 0;
}
```

运行实例：

Input a string: 123

Digit=123

【思考】运行以上程序时，如果输入 1#2#3，输出是什么？如果去掉程序中的 break 语句，同样输入 1#2#3，输出有变化吗？为什么？

【练习4.7】以下各项中不能正确将字符串存入一维字符数组的是____。

A. char s[10]="abcdefg";　　B. char t[]="abcdefg";

C. char s[10]; s="abcdefg";　　D. char s[10]={'a', 'b', 'c' ,'d', 'e', 'f', 'g'};

【练习 4.8】有以下程序：

```
#include<stdio.h>
int main()
{
    char s[]="\n12ab\\";
    printf("%d, %d\n", strlen(s), sizeof(s));
    return 0;
}
```

运行程序后的结果是_____。

A. 赋初值的字符串有错　　B. 输出 6, 7

C. 输出 5, 6　　D. 输出 6, 6

【练习 4.9】输入一个以回车结束的字符串(少于 80 个字符)，统计字符串中数字字符的个数。

4.3　二 维 数 组

4.3.1　二维数组的定义和引用

C 语言支持多维数组，其中最常见的是二维数组，本书只介绍二维数组，主要应用在表示二维表和矩阵中。

1. 二维数组的定义

二维数组定义的一般形式为：

类型说明符 数组名[常量表达式 1][常量表达式 2]；

数组名后必须是两个方括号括起来的常量表达式，各个常量表达式的值只能是正整数，常量表达式1 表示行长度，常量表达式 2 表示列长度。例如：

```
int a[2][3];        //定义一个二维数组 a，它有 2 行 3 列，共 6 个整型元素
float b[3][4];      //定义一个二维数组 b，它有 3 行 4 列，共 12 个 float 型元素
```

以上定义不能写成 int a[2, 3]; float b[3, 4];。

2. 二维数组元素的引用

引用二维数组的元素要指明两个下标，即行下标和列下标，一般形式为：

数组名[行下标][列下标]

行下标取值范围是从 0 开始，到行长度-1；列下标取值范围是从 0 开始，到列长度-1，引用时注意下标不要越界。

对前面定义的二维整型数组 a[2][3]，有两个行下标 0 和 1，三个列下标 0，1 和 2，6 个元素分为两行三列，第一行是：a[0][0]、a[0][1]、a[0][2]；第二行是：a[1][0]、a[1][1]、a[1][2]。可以表示一个 2 行 3 列的矩阵，如图 4.9 所示。

列下标		0	1	2
行下标	0	a[0][0]	a[0][1]	a[0][2]
	1	a[1][0]	a[1][1]	a[1][2]

图 4.9　二维数组表示的矩阵

3. 二维数组的存储顺序

二维数组可看作特殊的一维数组，数组每个元素又是一个包含若干元素的一维数组。如上述定义的数组 a[2][3]，可以看做是一维数组 a[2]，它当中每一个元素又是一个长度为 3 的一维数组。也就是说二维数组的每一行都可以看作是一个一维数组。

二维数组的元素在内存中按行优先的方式存放，即先存放 0 行的元素，再存放 1 行的元素……以此类推，其中每一行的元素再按照列顺序存放。前面定义的二维整型数组 a[2][3]，在内存中的存放顺序如图 4.10 所示。

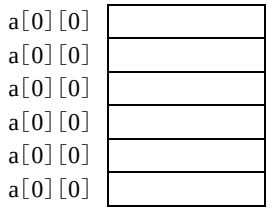


图 4.10　二维数组元素存储顺序

【注意】二维数组名 a 表示数组首元素的地址&a[0][0]；a[i]表示二维数组 i 行(第 i+1 行)首元素的地址&a[i][0]。

4.3.2　二维数组的初始化

二维数组也可以在定义时对各元素赋初值，初始化的方法有两种。

1. 将所有数据写在一对大括号内，按顺序赋值

例如：

```
int a[3][4]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
```

根据二维数组元素在内存中的存放顺序，把大括号“{”和“}”之间的数据依次赋给各元素。此时 a 数组中 3 行 4 列元素分别为：第一行：1，2，3，4；第二行：5，6，7，8；第三行：9，10，11，12。

对二维数组初始化时，也可以对部分元素赋初值，如“int a[3][4]={1, 2, 3, 4, 5, 6, 7, 8};”只对二维数组 a 的前两行 8 个元素赋了初值，第三行 4 个元素没赋初值，系统自动全赋为 0。

2. 分行给二维数组赋初值

例如：

```
int a[3][4]={{1, 2, 3, 4},{5, 6, 7, 8},{9, 10, 11, 12}};
```

二维数组的每一行又用一对大括号“{”和“}”括起来，等价于

```
int a[3][4]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
```

分行赋初值时也可以针对部分元素初始化，例如：

```
int a[3][4]={{1}, {0,6}, {0,0,11}};
```

等价于

```
int a[3][4]={1, 0, 0, 0, 0, 6, 0, 0, 0, 0, 11, 0};
```

初始化二维数组，如果是对全部元素都赋初值或分行赋初值，在初值表中列出了全部行，此时可以省略行长度，例如：

```
int a[][3]={1, 2, 3, 4, 5, 6, 7, 8, 9};
int b[][3]={{1,2,3}, {0}, {4,5}, {0}}
```

【注意】二维数　初始化时只能省略行长度，不能省略列长度，但建议不要省略行长度。

4.3.3　二维数组编程实例

常用二重 for 循环结构处理二维数组，由于二维数组的下标有行和列两部分，所以一般以行下标作为外层 for 循环的循环控制变量，以列下标作为内层循环的循环控制变量。

二维数组的应用最常见的是处理矩阵问题和英文文章问题。数值型二维数组，例如整型和浮点型二维数组主要用来处理矩阵问题，字符型二维数组主要用来处理英文文章问题。若有二维数组定义“int a[N][N];”，则用二维数组 a 表示的矩阵是 N 阶方阵，如图 4.11 所示，若用 i 和 j 分别表示 a 的行列下标，则其下标取值和 N 阶矩阵的特性之间有以下关系：

① 当 i==j 时，a[i][j]表示主对角线上的元素，图 4.11 中实斜线即矩阵主对角线。

② 当 i<=j 时，a[i][j]表示上三角中的元素，即图 4.11 中主对角线右上方部分。

③ 当 i>=j 时，a[i][j]表示下三角中的元素，即图 4.11 中主对角线左下方部分。

④ 当 i+j==N−1 时，a[i][j]表示副对角线上的元素，图 4.11 中虚斜线即矩阵副对角线。

$$\begin{pmatrix} a[0][0] & a[0][i] & a[0][N-1] \\ a[i][0] & a[i][i] & a[i][N-1] \\ a[N-1][0] & a[0][i] & a[i][N-1] \end{pmatrix}$$

【例 4.9】输入一个 2×3 的矩阵，要求编程输出该矩阵，并求出其中最小值以及最小值所在的行号和列号。

【分析】首先把第一个元素当成最小值，用二维数组元素 a[i][j]逐个与它比较，若比它小，则替换它，并记录行列下标。用 N-S 图表示的算法如图 4.12 所示。

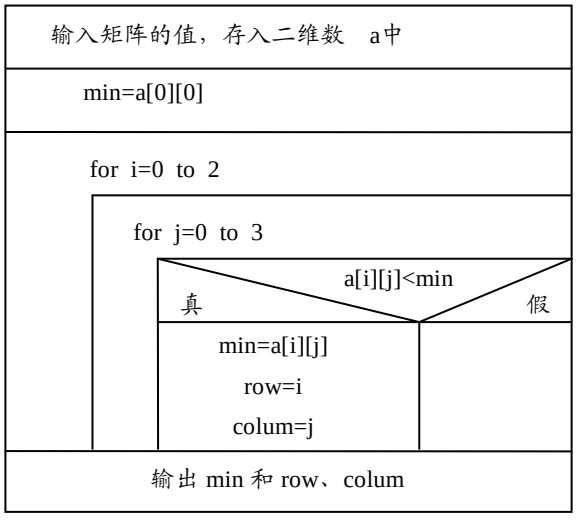


图 4.12　找矩阵元素最小值的 N-S 图

参考代码：

```
#include<stdio.h>
int  main()
{
    int  a[2][3];
    int  i, j, min, row=0, colum=0;
    /*将输入的数据存入二维数组 a*/
    printf("Enter  6  integers:\n");
    for(i=0; i<2; i++)
        for(j=0; j<3; j++)
            scanf("%d", &a[i][j]);
    /*按矩阵的形式输出二维数组 a*/
    for(i=0; i<2; i++)
    {
        for(j=0; j<3; j++)
            printf("%4d", a[i][j]);
        printf("\n");
    }
    /*遍历二维数组，找出最小值 a[row][colum]*/
    min=a[0][0];
    for(i=0; i<2; i++)
        for(j=0; j<3; j++)
            if(a[i][j]<min)
            {
                min=a[i][j];
                row=i;
                colum=j;
            }
    printf("min=%d, row=%d, colum=%d\n", min, row, colum);
    return  0;
}
```

运行实例：

```
Enter  6  integers:
-1  6  -9  3  2  10
   -1    6   -9
    3    2   10
min=-9, row=0, colum=2
```

【例4.10】编写程序，实现3行3列矩阵的转置(即行列互换)。

【分析】本程序可以分为输入矩阵、矩阵转置、输出矩阵三个部分。矩阵要用二维数组存放，所以用二维数组解决该问题。声明二维数组 a[3][3]存放 3 行 3 列矩阵，矩阵转置，就是将矩阵中 i 行 j 列的元素与 j 行 i 列的元素互换，即将 a[i][j]和 a[j][i]互换。

参考代码：

```
#include<stdio.h>
int  main()
{
    int  i, j, k;
    int  a[3][3];
    /*输入二维数组*/
    printf("输入 3 行 3 列矩阵：\n");
    for(i=0; i<3; i++)
        for(j=0; j<3; j++)
            scanf("%d", &a[i][j]);
    /*矩阵转置*/
    for(i=0; i<3; i++)
        for(j=0; j<3; j++)
          if(i<=j)
          {   k=a[i][j];
              a[i][j]=a[j][i];
              a[j][i]=k;
          }
    /*按矩阵形式输出二维数组*/
    printf("转置后矩阵：\n");
    for(i=0; i<3; i++)
    {
        for(j=0; j<3; j++)
            printf("%d ", a[i][j]);
        printf("\n");
    }
    return  0;
}
```

运行实例：

```
输入3行3列矩阵：
1  2  3
4  5  6
7  8  9
转置后矩阵：
```

```
1  4  7
2  5  8
3  6  9
```

二维数组还经常应用在处理多个字符串(英文文章)的问题中。

【例 4.11】用二维数组 str 存放五个字符串并输出，如图 4.13 所示。

参考代码段如下：

```
char str[][10] = {"Pascal", "Basic", "Fortran", "Java", "Visual C"};
for(i=0; i<5; i++)
    printf("%s\n", str[i]);
```

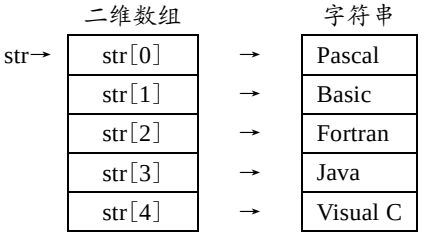


图4.13　用二维数组存放多个字符串

【例 4.12】输入一段 4 行的英文文章，每行少于 80 个字符，输出第二行的文字。

【分析】四行英文可以看做四个字符串，一个字符串可以用一维字符数组存储，多个字符串必须用二维字符数组存储，这样根据行下标才能区分行数。输入一行英文即输入一行字符串，所以四行要进行四次字符串的输入，用输入一行字符串的代码循环四次即可。

参考代码：

```
#include<stdio.h>
int main()
{   char str[4][80];
    int i;

    printf("输入四行英文：\n");
    for(i=0; i<4; i++)
        gets(str[i]);

    printf("输出第二行英文：\n");
        puts(str[1]);

    return 0;
}
```

运行实例：

```
输入四行英文：
How are you?
Fine, Thank you!
And you?
I am fine, too.
输出第二行英文：
Fine, Thank you!
```

【思考】用字符串的其他输入输出方式能否实现例 4.12 的要求？如何实现？

【提示】利用 scanf()函数的%s 格式符输入方式不能用来输入有空格的字符串，所以用%s 格式符输入方式实现不了处理英文文章问题。其他方式可以实现。

【练习4.10】以下语句中，能正确定义数组并正确赋初值的语句是_____。

A. int a[2][]={{1,2},{3,4}};　　B. int b[1][2]={{1},{3}};

C. int N=5, c[N][N];　　D. int d[3][2]={{1,2},{3,4}};

【练习4.11】有以下程序：

```
#include<stdio.h>
void main()
{
    int m[][3]={1, 4, 7, 2, 5, 8, 3, 6, 9};
    int i, j, k=2;
    for(i=0; i<3; i++)
        printf("%d ", m[k][i]);
}
```

运行程序后的输出结果是_____。

A. 4 5 6　　B. 2 5 8　　C. 3 6 9　　D. 7 8 9

【练习4.12】输入一个正整数n(1<n≤6)，根据下式生成一个n*n的方阵，然后将该方阵转置(行号列号互换)后输出。a[i][j] = i * n + j + 1(0≤i≤n−1，0≤j≤n−1)。

习　题　4

1. 编写程序，输入 5 个互异的整数存入一维数组 a 中，再输入一个整数 x，然后在数组中查找 x，如果找到输出相应的下标，否则输出“not found”。

2. 编写程序，输入一个正整数 n(1<n≤10)，再输入 n 个整数存入一维数组，把奇数从数组中删除，并统计出偶数个数。

3. 编写程序，输入一个正整数 n(1<n≤10)和 n 个有序整数(从小到大的顺序)，然后输入一个数 x，要求按原来排序的规律将它插入有序数据中，若原来序列中已有 x，则将原序列中的 x 删除。

4. 编写程序，输入一个以回车结束的字符串(少于80个字符)，再输入一个字符，统计并输出该字符在字符串中出现的次数，然后再输出该字符串。

5. 编写程序，输入一个以回车结束的字符串(少于 80 个字符)，将字符串中相同的字符存放在一起，并按 ASCII 码升序存放。

6. 编写程序，输入一个正整数n(1≤n≤10)，再输入n阶方阵a，计算该矩阵除副对角线、最后一列和最后一行以外的所有元素之和(副对角线为从矩阵的右上角到左下角的连线)。

7. 编写程序，输出一个九九乘法表。

【提示】用乘数、被乘数做下标，乘积放在一个二维数组中，再输出该二维数组。

第5章　函　　数

函数是用来实现某个特定功能的独立程序模块，它是C语言程序的基本组成单位。学习本课程以来所编写的每一个程序都用到了函数，如: main()函数以及标准输入输出函数scanf()和printf()等。一个完整的C程序由一个main()函数和若干个其他函数组成，C语言中所有语句都是以函数为载体的。

知识结构

1. 模块化程序设计
2. 函数的定义和调用
   ① 函数的定义
   ② 函数的调用
3. 变量的存储属性
   ① 自动变量
   ② 寄存器变量
   ③ 静态变量
   ④ 外部变量
4. 函数的嵌套调用
5. 递归函数
6. 数组作函数参数
   ① 数组元素作函数实参
   ② 一维数组名作函数参数
   ③ 二维数组名作函数参数

5.1　模块化程序设计

根据结构化程序设计的思想，一个较大的程序一般应分为若干个程序模块，每个模块再划分为更小的模块，每个小模块实现一个特定的功能。在C语言中，模块的功能由函数完成。一个C程序可由一个main()函数和若干个其他函数构成，main()函数用来解决整个问题，它调用解决小问题的其他函数，其他函数也可以相互调用。同一个函数可以被一个或多个函数调

用多次，可以说 C 程序的全部工作都是由各式各样的函数完成的。C 程序结构如图 5.1 所示。

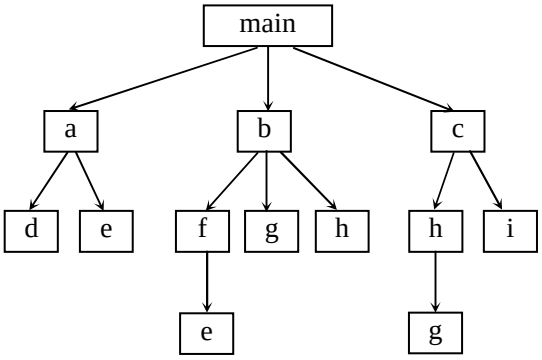


图 5.1　C 程序结构示意

1. 使用函数的主要作用

① 有利于增强程序的可读性。

如果将一个程序的代码全部写在 main()函数内，规模很大时，程序会越改越乱，有时会使得程序员连自己都看不明白自己编写的程序。通过定义子函数可以控制每一个函数的规模，因为各个函数功能相对独立，步骤有限，所以容易控制流程，编制和修改程序也比较容易。

② 有利于提高程序代码的复用性。

C 语言的库函数是系统提供的，是调试好的、常用的模块，我们可以直接利用。事实上我们自己编写的代码也可以重新利用，可以将已经调试好的，功能相对独立的程序代码段写成函数，供以后调用。这样可以大大减少编写重复代码的工作量，提高编程的效率。

③ 有利于多人分工协作完成程序开发。

将程序划分为若干函数，多个相对独立的函数可以由多人分别完成，每个人按照函数的功能要求和接口要求编制、调试代码，确保每个函数的正确性。

2. 函数分类

(1) 从用户使用的角度看，可分为库函数和用户自定义函数两种。

① 库函数：由 C 语言编译系统提供，无须用户定义，也不必在程序中作类型说明，只需在程序前包含有该函数原型的头文件即可在程序中直接调用。在前面各章的例题中反复用到的 printf()、scanf()、getchar()、putchar()、gets()、puts()、strlen()等函数均属此类。

② 用户自定义函数：由用户按需要写的函数。对于用户自定义函数，不仅要在程序中定义函数本身，而且在主调函数模块中还必须对该被调函数进行类型说明，然后才能调用。

(2) 从返回值的类型看，可分为有返回值函数和无返回值函数两种。

① 有返回值函数：此类函数被调用后向调用者返回一个执行结果，称为函数返回值。用户定义有返回值的函数时，必须在函数定义和函数声明中明确返回值的类型。

② 无返回值函数：此类函数用于完成某项特定的处理任务，调用后不向调用者返回函数值。用户定义此类函数时可指定它为“空类型”，空类型的说明符为“void”。

(3) 从函数的形式看，可分为无参函数和有参函数两种。

① 无参函数：函数定义、函数声明及函数调用中均不带参数。主调函数和被调函数之间不进行参数传递。

② 有参函数：函数定义及函数声明时都有参数，称为形式参数(简称为形参)。在函数调用时也必须给出参数，称为实际参数(简称为实参)。进行函数调用时，主调函数将把实参的

值传递给形参，供被调函数使用。

应该指出的是，main()函数是主函数，它可以调用其他函数，而不允许被其他函数调用。因此，C 程序的执行总是从 main()函数开始，完成对其他函数的调用后再返回到 main()函数，最后由 main()函数结束整个程序。一个 C 源程序必须有、而且只能有一个 main()函数。

在 C 语言中，所有的函数定义，包括主函数 main()在内，都是平行的。也就是说，在一个函数的函数体内，不能再定义另一个函数，即函数不能嵌套定义。但是函数之间允许相互调用，也允许嵌套调用。习惯上把调用者称为主调函数，被调用者称为被调函数。函数还可以自己调用自己，称为递归调用。

有时候为了避免一个文件过长，可以把一个程序分别保存成几个文件。这样一个大程序可能由几个文件组成，每个文件又可能包含若干个函数。这种保存有一部分程序的文件称为程序文件模块。

5.2 函数的定义和调用

5.2.1 函数的定义

函数是一个完成特定工作的独立程序模块，程序中一旦调用了某个函数，该函数就会完成一些特定的工作，然后返回到调用它的地方。这时有两种情况：

① 调用函数后得到一个明确的运算结果，并需要将该值返回到主调函数。

② 调用函数只是完成了一系列操作步骤，不返回任何值。

1. 有返回值的函数定义

有返回值函数定义的一般形式如下：

```
函数类型 函数名(形参表)              //函数首部
{                                    //函数体
    声明部分;
    语句部分;
    return 表达式;
}
```

一个函数由函数首部和函数体两部分组成。

① 函数首部：函数首部包括函数的类型、函数的名称以及形参表三个部分。

❖ 函数类型—— 函数返回值的数据类型，可以是基本数据类型也可以是构造类型。如果省略，默认为 int 类型。

❖ 函数名—— 给函数取的名字，以后用这个名字调用该函数。函数名由用户命名，需符合自定义标识符的命名规则。

❖ 形参表—— 函数名后面“()”号里包含形式参数，简称形参。形参表说明各形式参数的类型和名称，各个形式参数间用“,”分隔。在调用函数时，主调函数将赋予这些形式参数实际的值。无参函数没有参数传递，但“()”号不能省略。

② 函数体：函数首部后面用一对“{}”括起来的部分是函数体。如果函数体内有多个“{}”，最外层包含的内容是函数体的范围。

函数体由声明部分、语句部分和 return 语句组成。

❖ 声明部分——定义本函数所使用的变量和进行有关声明(如后面要讲的函数声明)。

❖ 语句部分——由若干条语句组成的命令序列(可以在其中调用其他函数)。

❖ return 语句：函数返回值由 return 返回。

【例 5.1】定义一个用于判断奇偶数的函数，当为偶数时返回 1，否则返回 0。

【分析】根据函数定义的形式，分析出函数的首部和函数体即可写出函数。

函数首部：函数类型——函数返回值的类型是 int 型；函数名——even；形参表——需要一个参数且是 int 类型。

函数体：实现判断奇偶功能，很明显，应该是二分支结构。

参考代码：

```
int even(int n)                    // 函数首部
{                                  // 函数体
    if(n%2 == 0)                   // 判别奇偶数
        return 1;                  // 如果是偶数，返回 1
    else
        return 0;                  // 如果是奇数，返回 0
}
```

第一行第一个关键字 int 说明本函数是一个整型函数，其返回值是一个整数。even 是函数名。形参 n 为整型变量，n 的具体值是由主调函数在调用时传送过来的。在“{}”中的函数体内，除形参外没有使用其他变量，因此只有语句而没有声明部分。函数体中的 return 语句把 1 或 0 作为函数的值返回给主调函数。有返回值的函数中至少应有一个 return 语句。

函数的返回值是指函数被调用之后，执行函数体中的程序段所取得的并返回给主调函数的值。如调用例 5.1 的 even()函数取得的 1 或 0。

【说明】① 函数的返回值只能通过 return 语句返回主调函数。return 语句的一般形式为:

```
return 表达式;
```

或者为:

```
return (表达式);
```

该语句的功能是计算表达式的值，并返回给主调函数。在函数中允许有多个 return 语句，但每次调用只能有一个 return 语句被执行，因此只能返回一个函数值。

② 函数返回值的类型和函数定义中的函数类型应保持一致。如果两者不一致，则以定义中的函数类型为准，自动进行类型转换。

③ 如果函数返回值为整型，在函数定义时可以省去类型说明。

2. 无返回值的函数定义

无返回值函数定义的一般形式如下：

```
void 函数名(形参表)          //函数首部
{                            //函数体
    声明部分
    语句部分
}
```

函数类型为 void，表示无返回值，函数体中无 return 语句。void 类型的函数不直接返回一个值，它的作用通常以屏幕输出等方式体现。

【例 5.2】编写函数输出 n 行星号，例如 n 为 4 时的输出如下所示。

```
   *
  * *
 * * *
* * * *
```

【分析】函数功能就是输出 n 行星号，不做任何运算，也没有任何运算结果，自然也不需要返回值，因此函数类型应为 void 类型。形参应为整型变量，用来决定星号的行数。

参考代码：

```
void star(int n)
{
    int i, j;
    for(i=1; i<=n; i++)
    {
        for(j=1; j<=n-i; j++)
            printf(" ");
        for(j=1; j<=i; j++)
            printf("* ");
        printf("\n");
    }
}
```

5.2.2 函数的调用

在程序中通过调用函数来执行函数体，调用函数时，将实参值传递给形参并执行函数定义中的语句，以实现相应的功能。

1. 调用函数的一般形式

调用函数的一般形式为：

函数名(实参表)

实参表中的参数可以是常量、变量或其他构造类型的数据及表达式，若有多个实参，各实参之间用逗号分隔。例如，

```
star(4);          //调用例 5.2 中 star 函数
```

【例 5.3】输入 1 个整数，判断该数是奇数还是偶数。

【分析】判断奇偶数的函数在例 5.1 中已经定义，根据调用函数的形式，编写调用语句(见下面程序代码中加粗显示的内容)。

参考代码：

```
#include<stdio.h>
int even(int n)                    //函数首部
{
    if(n%2 == 0)                   //判别奇偶数
        return 1;                  // 如果是偶数，返回 1
    else
        return 0;                  // 如果是奇数，返回 0
```

```
}
int main(void)
{
    int number;
    printf("Enter a number: ");
    scanf("%d", &number);
    if(even(number) == 1)          //调用 even 函数，根据调用结果判断奇偶数
        printf("The number is even. \n");
    else
        printf("The number is odd. \n");
    return 0;
}
```

主函数调用例 5.1 中编写的判断奇偶数的函数 even(number)，其中 even 是函数名，number 作为实参。因为 even 函数的返回值有两种情况 1 或 0，所以在主函数中用 if-else 语句根据函数调用结果判断 number 是奇数还是偶数。

2. 调用函数的过程

主函数是每个程序的起始执行点，对于一个 C 程序，不论 main()函数在程序中的位置如何，总是从 main()函数开始执行。如果遇到函数调用语句，主函数暂停执行，转而执行被调用的函数，执行完该函数后，再返回主函数，然后从原先暂停的位置继续向下执行。

下面以例 5.3 为例分析函数调用的过程。

在 main()函数中，当运行到“if(even(number) == 1)”时，暂停 main()函数的执行，调用 even()函数。将实参 number 的值传给形参 n，并执行 even()函数中的语句，当执行到 return 语句时，结束函数调用，将函数的返回值带回到主函数中调用它的地方继续向下执行，根据调用结果判断输出 number 是奇数还是偶数。

对于 void 类型的函数，如例 5.2 中定义的 star()函数没有 return 语句，执行完函数中所有语句后，遇到最后的大括号自动返回主调函数。

3. 函数参数的传递

函数的参数分为形式参数(形参)和实际参数(实参)两种，函数定义中的参数称为形参，调用函数时的参数称为实参。当执行程序遇到调用函数时，主调函数把实参的值传送给被调函数的形参，即参数传递，从而实现主调函数向被调函数的数据传送。

如例 5.3 函数定义“int even(int n)”中定义的 n 是一个形参。而主函数 main()中通过“if(even(number)==1)”调用函数时给出的 number 是实参。调用函数时，实参 number 的值将被传递给形参 n。

函数的形参和实参具有以下特点：

① 形参必须是变量，用于接收实参传递过来的值，它的使用方法和普通变量相同；实参可以是常量、变量、表达式、函数调用等，无论实参是何种类型的量，在调用函数时，它们都必须具有确定的值，以便把这些值传递给形参。

② 如果实参是变量，它与所对应的形参是两个不同的变量。实参是主调函数的变量，形参是被调函数的变量，两者可以同名也可以不同名。

③ 实参和形参在数量、类型、顺序上必须一一对应，否则会发生“类型不匹配”的错误。

④ 在参数传递过程中，实参把值复制给形参。这种参数传递是单向的，即只能把实参的

值传送给形参，而不能把形参的值反向传送给实参。因此在函数调用过程中，如果形参的值改变了不会影响实参。

【例 5.4】形参的值改变了不会影响实参的示例。

参考代码：

```
#include<stdio.h>
int  main(void)
{
    int  a=3, b=4;
    void  swap(int  x, int  y);          //函数声明
    swap(a, b);                          //调用 swap 函数
    printf("a=%d, b=%d\n", a, b);
    return  0;
}
void  swap(int x,int y)                  //定义 swap 函数，其作用是若 x<y，则交换 x 和 y 的值
{
     int  t;                             //定义中间变量 t
     if(x<y)
     {
          t=x;
          x=y;
          y=t;
     }
     printf("x=%d, y=%d\n", x, y);
}
```

运行实例：

```
x=4, y=3
a=3, b=4
```

执行程序时，主函数中调用 swap(a,b)函数，将实参 a、b 的值 3 和 4 分别传给形参 x 和 y，在执行函数过程中，形参 x 和 y 的值发生了交换，因此输出 x=4，y=3。返回主函数后，输出实参 a 和 b 的值，仍为 3 和 4，即 a=3，b=4。可见实参的值不随形参的变化而变化，因为在内存中实参变量与形参变量占用不同的单元，如图 5.2 所示。

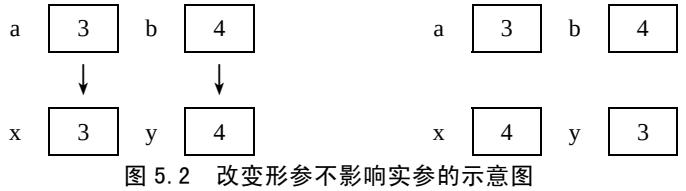


图 5.2　改变形参不影响实参的示意图

4. 函数的声明

C 语言要求函数必须先定义后调用，将被调函数放在主调函数的前面，就像变量必须先定义后使用一样。如果被调函数的定义放在主调函数定义的后面，就需要在函数调用前加上函数声明。

函数声明的目的是把函数类型、函数名和函数参数的类型、个数等信息通知编译系统，以便在遇到函数调用时，编译系统能正确识别函数并检查调用是否合法。

函数声明的一般形式为：

函数类型 函数名(类型 形参, 类型 形参…)；

或为：

函数类型 函数名(类型, 类型 …)；

括号内给出了形参的类型和形参名，或只给出形参类型，因为编译系统只检查参数类型而不检查参数名。

如例 5.4 中，在 main()函数内部对 swap()函数的声明为：“void swap(int x, int y);”也可以写为：“void swap(int , int);”。

【说明】① 函数声明由函数定义的第一行，即函数首部加上一个分号构成。因为函数声明是一条语句，因此要以分号结束；而定义函数时，函数首部不是语句，故后面不能加分号。

② 如果被调函数的定义出现在主调函数之前，在主调函数中可以不对被调函数进行声明而直接调用。如例 5.3 中，even()函数的定义放在 main()函数之前，因此在 main()函数中可以省去对 even()函数的声明。

③ 调用库函数时不需要进行函数声明，但必须在程序文件开头用#include 命令把该函数的头文件包含到本文件中，例如使用输入输出函数时需用到#include<stdio.h>。

【例 5.5】求 1～n(n<=200)之间的全部素数，每行输出 10 个。注意，1 不是素数，2 是素数。要求定义和调用函数 prime(m)判断 m 是否为素数，当 m 为素数时返回 1，否则返回 0。

【分析】本题需要对 1～n 之间的每一个数进行判断，如果是素数就输出，可以用下述循环来实现。

```
for(m = 1; m <= n; m++)
    if(m 是素数)
        输出 m ;
```

第 3 章中已经介绍了判断素数的方法，也可以判断 m 能否被 2～$\sqrt{m}$ (或 2～m/2)之间的数整除，如果能被其中任何一个数整除，则 m 不是素数，否则是素数。本题要求将判断素数的过程用自定义函数来实现，函数返回值为 int 型，当 m 为素数时返回 1，否则返回 0。

参考代码：

```
#include<stdio.h>
#include<math.h>                    // 本题调用求平方根函数 sqrt()，需要包含 math.h
int  main(void)
{
    int  count, m, n;
    int  prime(int  m);             //函数声明
    count = 0;
    printf("Enter  n:");
    scanf("%d", &n);
    printf("The  prime  between  1  to  %d:\n", n);
    for(m = 1; m <= n; m++)
    {
        if(prime(m))                        //调用 prime 函数判断 m 是否为素数
```

```
            {
                printf("%4d", m);          // 输出 m
                count++;                   // 累加已经输出的素数个数
                if(count %10 == 0)
                   printf("\n");           //如果 count 是 10 的倍数，换行
             }
         }
         printf("\n");
         return  0;
    }
    //定义判断素数的函数，如果 m 是素数则返回 1("真")；否则返回 0("假")
    int  prime(int  m)
    {
        int  i, n;
        if(m == 1)      return 0;          // 1 不是素数，返回 0
          n = sqrt(m);
          for(i = 2; i <= n; i++)
              if(m % i == 0)               // 如果 m 不是素数
              {
                  return  0;               // 返回 0
              }
          return  1;                             //m 是素数，返回 1
    }
```

运行实例：

```
Enter  n: 100
The  prime  between  1  to 100:
   2    3    5    7   11   13   17   19   23   29
  31   37   41   43   47   53   59   61   67   71
  73   79   83   89   97
```

【练习5.1】有如下函数定义：

```
void  fun(int  n, double  x) {.....}
```

若以下选项中的变量都已正确定义并赋值，则正确调用fun函数的语句是_____。

A. fun(int y, double m);　　B. k=fun(10, 12.5);

C. fun(x, n);　　D. void fun(n, x);

【练习5.2】有以下程序：

```
#include<stdio.h>
fun(int  a, int  b)
{
    if(a>b)
        return(a);
    else
```

```
        return(b);
    }
    void  main()
    {
        int  x=3, y=8, z=6, r;
        r=fun(fun(x,y), 2*z);
        printf("%d", r);
    }
```

运行程序后的输出结果是_____。

A. 3　　B. 6　　C. 8　　D. 12

【练习 5.3】有以下程序：

```
    #include<stdio.h>
    void  f(int  x, int  y)
    {
        int  t;
        if(x<y){ t=x;    x=y;    y=t; }
    }
    void  main( )
    {
        int  a=4, b=3, c=5;
        f(a, b);  f(a, c);  f(b, c);
        printf("%d, %d, %d\n", a, b, c);
    }
```

运行程序后的输出结果是_____。

A. 3, 4, 5　　B. 5, 3, 4　　C. 5, 4, 3　　D. 4, 3, 5

【练习 5.4】输入两个正整数，输出其中较大的数。要求通过定义和调用 max(x, y)函数求 x 和 y 中较大的数。

【练习 5.5】输入两个正整数 m 和 n(m≥1，n≤300，m<n)，求 m 和 n 之间所有素数的和。要求通过定义和调用 prime(p)函数判断 p 是否为素数，当 p 为素数时返回 1，否则返回 0。

5.3　变量的存储属性

变量是对程序中数据存储空间的抽象。变量除了具有数据类型特性外，还有存储属性特性。在 C 语言中，变量的存储属性体现在以下三个方面。

① 按变量存储位置区分，有位于寄存器的变量，有位于主存的变量，不同存储位置的变量，访问速度及方式也不同。

② 按变量存在的时间(即生存期)及分配方式区分，有静态存储方式和动态存储方式。静态存储的变量在编译时分配存储位置，在程序运行时有固定的存储空间。动态存储的变量在程序运行期间根据需要进行动态的存储空间分配。

③ 按变量的作用范围(即作用域)不同区分，可分为局部变量和全局变量两种。

局部变量也称为内部变量，它是定义在函数内或复合语句内的变量，其作用域仅限于定义它的函数或复合语句内部，离开该函数或复合语句后再使用这种变量是非法的。例如，形参就是局部变量，只在它们所在的函数范围内有效。迄今为止，我们所接触的变量，都是局部变量。

全局变量也称为外部变量，它是在函数外部定义的变量。它不属于哪一个函数，而属于整个源程序文件。它的作用范围是从变量定义处开始，到本程序文件结束，它对作用范围内所有的函数都起作用。

在 C 语言中用 4 个类型说明符 register，auto，static，extern 来表示变量的存储属性。

在定义一个变量时，除了指定其数据类型外，还可以指定其存储属性，一般定义格式为：

存储类型 数据类型 变量表;

如“register int a;”语句定义了一个寄存器类型的整型变量 a。

5.3.1 自动(auto)变量

自动(auto)变量存储方式是 C 语言默认的局部变量的存储方式，也是局部变量最常用的存储方式，其作用域仅限于定义它的函数或复合语句内。

程序执行时从主函数开始，主函数中的局部变量一开始就在内存中分配了存储单元。而其他函数在被调用之前，函数中定义的局部变量并没有分配内存单元，只有在调用函数时，才为其中的局部变量分配相应的内存单元；一旦函数调用结束，被调函数中定义的所有局部变量将不复存在，相应的内存单元由系统自动收回。根据这种特性，将这类局部变量称为自动变量，即函数被调用时，系统自动为其局部变量分配内存单元；函数调用结束，系统自动收回所分配的内存单元。

变量从分配内存单元开始，到内存单元被收回的整个过程，称为变量的生命周期。变量的生命周期和作用范围是两个不同的概念。

自动变量定义的一般形式为：

auto 类型名 变量名表;

其中 auto 为自动存储类别关键词，可以省略，省略时，系统默认变量类型为 auto 型。例如“auto int a, b;”语句与“int a, b;”语句完全等价，它们都定义了两个 auto 型整型变量 a、b。前面介绍的函数中定义的变量虽然都没有声明为 auto 型，其实都隐含指定为 auto 型，也就是说前面程序中定义的局部变量都是自动变量。

【说明】① 主函数中定义的局部变量也只在主函数中有效，不能在其他函数中使用。同样，主函数中也不能使用其他函数中定义的变量。因为主函数也是一个函数，它与其他函数是平行关系。

② 不同函数中可以使用相同的局部变量名，它们代表不同的对象，占有不同的内存单元，互不干扰，不会发生混淆。

③ 除了作用于函数的局部变量外，还可以定义作用于复合语句的局部变量，这类变量只在复合语句范围内有效。

【例 5.6】复合语句中局部变量应用示例。

参考代码：

```
#include<stdio.h>
int main(void)
```

```
{
    int  i=2, j=3, k;              // 局部变量 i，j，k 在主函数内有效
    k=i+j;                         // k=5
    {                              //复合语句开始
        int  k=8;                  //局部变量 k 的作用范围被局限在该复合语句内
        printf("%d\n", k);
    }                              //复合语句结束
    printf("%d\n", k);
    return  0;
}
```

运行实例：

```
8
5
```

本程序在 main 函数中定义了 i，j，k 三个变量，并给 k 赋 i+j(为 5)的值。而在复合语句内又定义了一个变量 k，并赋初值为 8。应该注意，这两个 k 不是同一个变量。因此，在复合语句内输出 k 的值是 8，在复合语句外输出 k 的值是 5。

【注意】函数中的局部变量和复合语句中的局部变量作用范围不同，可以同名。当两者同名时，以复合语句的局部变量优先，即在复合语句范围内，函数局部变量不起作用。

使用局部变量可以避免各个函数之间的变量相互干扰。如例 5.4 中，在 main()函数内部有局部变量 a 和 b(实参)，在 swap()函数内部有局部变量 x 和 y(形参)。它们是定义在不同函数内的局部变量，各有自己的存储单元和作用范围，相互之间不会产生干扰，所以形参 x 和 y 的改变不会影响实参 a 和 b 的值。

5.3.2　寄存器(register)变量

寄存器(register)变量存储方式是 C 语言中使用较少的一种局部变量的存储方式。该方式将局部变量存储在 CPU 的寄存器中，计算机对寄存器的操作比对内存要快很多，所以可以将程序中在一段时间内要反复使用的局部变量存放在寄存器中。使用寄存器变量的目的就是为了加快程序运行速度。

寄存器变量只适用于整型，其定义的一般形式为：

```
register  int  变量名表;
```

寄存器变量的使用方式与自动变量相同，一般情况下较少使用。

5.3.3　静态(static)变量

静态(static)变量的存储方式是在程序运行期间分配固定存储空间的方式。

静态变量定义的一般形式为：

```
static  类型名  变量名表;
```

C 语言中，使用静态存储方式存储的变量主要有静态局部变量和全局变量。

1. 静态局部变量

静态局部变量的存储空间在程序编译时由系统分配，在整个程序运行期间其空间位置都固定不变。该类变量在包含它的函数调用结束后仍然保留变量值。下次调用该函数，静态局部变量中仍保留上次调用结束时的值。

静态局部变量的初值在编译程序时一次性赋予，在程序运行期间不再执行赋初值语句，以后若改变了值，则保留最后一次改变后的值，直到程序运行结束。对未赋初值的静态局部变量，C 编译程序自动给它赋初值 0。

【例 5.7】静态局部变量应用示例。

参考代码：

```
#include<stdio.h>
int  main(void)
{
     int  k;
     void  fun(int  k);
     for(k=1; k<4; k++)
          fun(k);              // 循环调用 3 次
     printf("\n");
     return  0;
}
void  fun(int  k)
{
     static  int  x;              // 静态局部变量 x 的初值为 0
     printf("%d ", x);
     x+= k ;                      // 静态局部变量 x 会记住前一次调用时留下来的值
}
```

运行实例：

```
0 1 3
```

例 5.7 中，主函数循环调用 3 次 fun()函数，每一次进入 fun()函数，静态局部变量 x 都会记住上次调用留下的值。

【说明】① 对自动变量，在函数调用时分配内存单元，函数调用结束后系统自动收回该内存单元，下一次调用时再重新分配。而静态局部变量在函数第一次被调用时分配内存单元，之后在整个程序运行期间都不释放。

② 自动变量如果定义时不赋初值，它的值是一个随机值。而静态局部变量如果定义时不赋初值，系统自动赋初值 0。

③ 自动变量赋初值是在函数调用时进行，每调用一次函数重新赋一次初值，相当于执行一次赋值语句。而静态局部变量只在函数第一次调用时赋初值，即只赋初值一次，以后调用都按前一次保留的值使用。

2. 全局变量

前面已经提到过全局变量的作用范围，使用全局变量可以解决多个函数间的变量共用问题，便于函数之间进行数据交流。

全局变量和局部变量的作用范围不同，可以同名。当两者同名时，在对应的函数中全局变量被屏蔽不起作用，由局部变量起作用。对于其他不存在同名变量的函数，全局变量仍然有效。

【例 5.8】考察全局变量的作用范围的应用示例。

参考代码：

```
#include<stdio.h>
int  a=3, b=5;                    //定义全局变量 a，b
int  max(int  a, int  b)
{
     int  c;
     c=a>b?a:b;
     return  c;
}
int  main(void)
{
     int  x, b = 3;               // 定义局部变量 x，b，在本函数中，全局变量 b 不起作用
     x = a;                       // 对变量 x 赋值 3
     {
          int  b = 4;
          b = x + b;              //b 的值为 7
          x = max(a, b);          //a 为全局变量，b 为复合语句中的局部变量
     }
     printf("a=%d, b=%d, x=%d\n" , a, b, x );
     return  0;
}
```

运行实例：

```
a=3, b=3, x=7
```

在 C 语言中，全局变量都采用静态存储方式存储，即在编译时就为相应全局变量分配固定的存储单元，且在程序执行全过程中存储位置保持不变。对全局变量赋初值也是在编译时完成的。

因为全局变量全部是静态存储，所以没有必要为说明全局变量是静态存储而使用关键词 static。如果使用了 static 说明全局变量，并不是说明“此全局变量要用静态方式存储”（因为全局变量本来就是静态存储的）；而是说，这个全局变量只在本程序文件模块有效，不能被其他的程序文件模块引用。

例如，一个程序由两个程序文件 file1.c 和 file2.c 组成，在 file1.c 中定义了全局变量 x，并用 static 声明，则 x 只能用于程序文件 file1.c 中，虽然在 file2.c 对 x 作 extern 声明，也无法使用 file1.c 中的 x，如图 5.3 所示。

文件名：file1.c

```
static  int  x;
void  main()
{
   ……
}
```

文件名：file2.c

```
extern  x;
//无法使用file1.c中的全局变量x
int  f1()
{
    ……
}
```

图 5.3　**static** 声明限制全局变量的作用范围

5.3.4　用 extern 声明外部变量

全局变量的作用范围是从变量定义处开始，到本程序文件结束。编译时将全部变量分配在静态存储区，有时需要用 extern 来声明外部变量，以扩展全局变量的作用域。

1. 在一个文件内声明外部变量

全局变量一般定义在程序文件的开头，这样它对整个程序文件内的所有函数都有效。如果全局变量不定义在文件的开头，它的有效作用范围只限于定义处开始到文件结束。如果在定义点之前的函数想引用该全局变量，就应该在引用之前用关键字 extern 对该变量作“外部变量声明”，以扩展全局变量的作用范围。有了外部变量声明，就可以从“声明”的地方开始，合法地使用该全局变量。

外部变量声明的一般形式如下：

extern(类型名) 变量名表;

其中类型名可以省略。外部变量声明只起说明作用，不分配内存单元，对应的内存单元在全局变量定义时分配。

【例 5.9】用 extern 声明外部变量，扩展全局变量在程序文件中的作用范围示例。

参考代码：

```
#include<stdio.h>
int  max(int  x, int  y)          //定义 max 函数
{
     int  z;
     z=x>y?x:y;
     return(z);
}
int  main(void)
{
    extern  a, b;                 //外部变量声明
    printf("max=%d", max(a, b));
}
int  a=13, b=23;
```

运行实例：

```
max=23
```

例 5.9 中的程序文件在最后一行定义了全局变量 a，b，由于全局变量定义的位置在函数 main()之后，因此在 main()函数中本来不能引用全局变量 a 和 b。现在在 main()函数的第 2 行用 extern 对 a 和 b 进行“外部变量声明”，表示 a 和 b 是已经定义的外部变量(但定义的位置在后面)。这样在 main()函数中就可以从声明的地方开始，合法地使用全局变量 a 和 b 了。如果不作 extern 声明，编译时将出错，系统不会认为 a、b 是已定义的外部变量。实际上较好的做法是把外部变量的定义放在引用它的所有函数之前。

2. 在多文件的程序中声明外部变量

如果一个程序由多个程序文件模块组成，也可以通过外部变量声明，使全局变量的作用范围扩展到其他程序文件模块。

例如，一个程序由两个程序文件 file1.c 和 file2.c 组成，在 file1.c 中定义了全局变量 x，在 file2.c 对 x 作 extern 声明后，就可以合法使用 file1.c 中的 x，如图 5.4 所示。

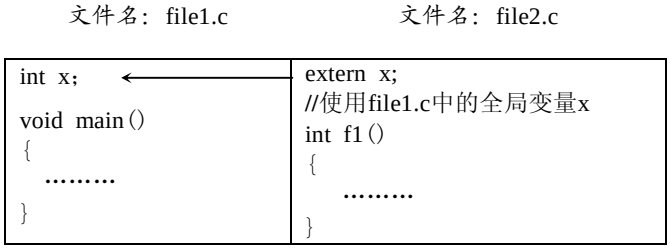


图 5.4　用 **extern** 声明扩展全局变量的作用范围

【总结】普通全局变量的作用范围从定义处开始，到整个程序文件结束；用 extern 声明外部变量，可以扩展全局变量的作用范围；静态全局变量的作用范围是所在的程序文件模块，它是小范围的全局变量；函数的局部变量作用范围是所在的函数；复合语句的局部变量作用范围仅在复合语句内，范围最小。

【练习5.6】有以下程序：

```
#include<stdio.h>
int m=13;
int fun2(int x, int y)
{
    int m=3;
    return(x*y-m);
}
void main( )
{
    int a=7, b=5;
    printf("%d\n", fun2(a, b)/m);
}
```

运行程序后的输出结果是_____。

A. 1　　B. 2　　C. 7　　D. 10

【练习5.7】有以下程序：

```
#include<stdio.h>
int fun()
{
    static int x=1;
    x*=2;
    return x;
}
void main()
{
    int i, s=1;
    for(i=1; i<=2; i++)
    s=fun();
    printf("%d", s);
```

```
}
```

运行程序后的输出结果是_____。

A. 0　　B. 1　　C. 4　　D. 8

5.4　函数的嵌套调用

C 语言中各函数之间是平行的、独立的，函数间不存在包含关系，不允许出现函数的嵌套定义。但允许在一个函数的定义中出现对另一个函数的调用，即在调用一个函数过程中又调用另一个函数，这就构成了函数的嵌套调用，其关系可如图 5.5 所示。

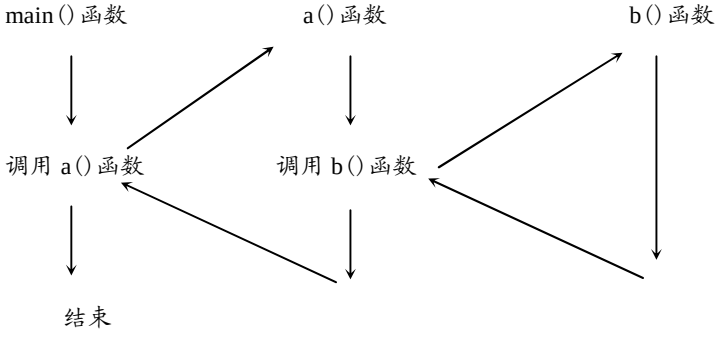


图 5.5　函数嵌套调用

图 5.5 表示了两层嵌套的情形。其执行过程是：执行 main()函数中调用 a()函数的语句时，即转去执行 a()函数，在 a()函数中调用 b()函数时，又转去执行 b()函数，b()函数执行完毕返回 a()函数的断点继续执行，a()函数执行完毕返回 main()函数的断点继续执行。

【例 5.10】输入两个正整数 m 和 n，编写程序计算并输出它们的最小公倍数和最大公约数。要求分别定义和调用 f1(m, n)函数计算最小公倍数，调用 f2(m, n)函数计算最大公约数。

【分析】要求两个数的最小公倍数和最大公约数，首先我们要明确最小公倍数和最大公约数的关系：最大公约数×最小公倍数=两数乘积，所以只要求出其中一个，另一个很容易求出。本题定义 f1()函数，求 m 和 n 的最小公倍数；定义 f2()函数，求两数的最大公约数。在主函数中分别调用这两个函数，并输出函数调用的结果。其中调用函数 f2()的过程中又需要调用函数 f1()，从而构成函数的嵌套调用。函数调用过程如图 5.6 所示。

参考代码：

```
#include<stdio.h>
int  main(void)
{
    int  n, m;
    int  f1(int  m, int  n);   //函数声明
    int  f2(int  m, int  n);   //函数声明

    printf("Input  m:");
    scanf("%d", &m);
    printf("Input  n:");
    scanf("%d", &n);
```

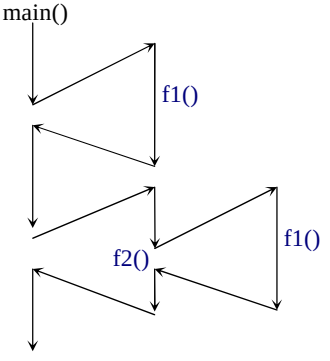


图 5.6　函数调用过程

```
        printf("最小公倍数是%d\n", f1(m, n));    //函数调用
        printf("最大公约数是%d\n", f2(m, n)); //函数调用
        return 0;
    }
    int f1(int m, int n)            //函数定义，求最小公倍数
    {
        int j;
        j=m;
        while(j%n!=0)
          j=j+m;
        return j;
    }
    int f2(int m, int n)            //函数定义，求最大公约数
    {
        int k;
        k=m*n/f1(m, n);             //函数调用
        return k;
    }
```

运行实例 1：

```
Input m: 3
Input n: 7
最小公倍数是 21
最大公约数是 1
```

运行实例 2：

```
Input m: 24
Input n: 60
最小公倍数是 120
最大公约数是 12
```

5.5　递 归 函 数

在调用一个函数的过程中又直接或间接地调用该函数本身，称为函数的递归调用，带有递归调用的函数也称为递归函数。

例如有函数 f() 如下：

```
int f(int x)
{
    int y;
    ……
    y=f(x);
    ……
    return y;
}
```

本例在 f() 函数中又调用 f() 函数，这是直接递归调用。如果在 f() 函数中调用 g() 函数，

而在 g()函数中又调用 f()函数，就是间接递归调用。如图 5.7 所示。

函数直接递归调用

```
int f(int x)
{
    int y;
    ……
    y=f(x) ;
    ……
    return y;
}
```

函数间接递归调用

```
int f(int x)
{
    int y;
    ……
    y=g(x) ;
    ……
    return y;
}
```

```
int g(int x)
{
    int z;
    ……
    z=f(x) ;
    ……
    return z;
}
```

图 5.7　函数递归调用的两种形式

图 5.7 所示的这两种递归调用都是无终止的自身调用，这显然不正确。为了防止递归调用无终止地进行，常用的办法是用 if 语句来控制，只有满足某一条件时才执行递归调用，否则不再继续调用。下面举例说明递归调用的执行过程。

【例 5.11】用递归法计算 n!

【分析】n!等于(n-1)!乘以 n，(n-1)!又等于(n-2)!乘以 n-1……2!等于 2 乘以 1!，1!等于 1，0!等于 1。用递归法计算 n!可用下述公式表示：

$$n!=\begin{cases}1 & (n=0,1)\\ n\times(n-1)! & (n>1)\end{cases}$$

参考代码：

```
#include<stdio.h>
double fact(int n);                          //函数声明
int main(void)
{
    int n;

    printf("Input n: ");
    scanf("%d", &n);
    printf("%d!=%.0f\n", n, fact(n) );       //函数调用
    return 0;
}
double fact(int n)                           //函数定义
{
    double f;
    if (n==1 || n==0)                        //递归出口
       f = 1;
    else
       f = n * fact(n-1);                    //函数递归调用
    return f;
}
```

运行实例：

```
Input n: 3
```

3!=6

例 5.11 程序中定义了递归函数 fact()，主函数调用 fact()后，进入函数 fact()，如果 n=0 或 n=1 时都将结束函数的执行，否则就递归调用 fact()函数自身。

下面我们以 fact(3)为例来分析递归函数的执行过程，如图 5.8 所示。

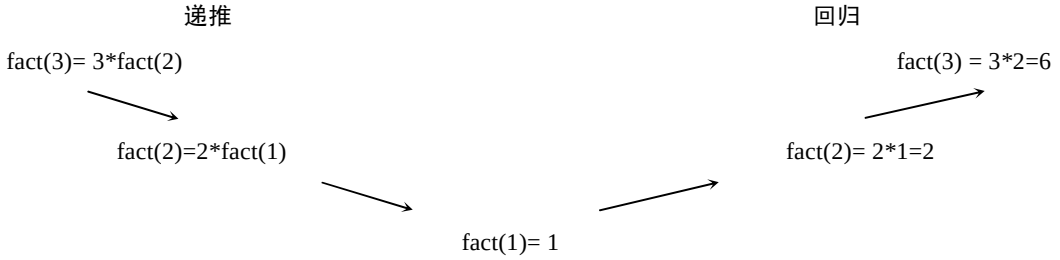


图 5.8　fact(3)递归函数执行过程

由图 5.8 可看出，求解可分成两个阶段：第一阶段是“递推”，即将 n 的阶乘表示成(n−1)的阶乘的函数，而(n−1)的阶乘还要递推到(n−2)的阶乘……直到 1 的阶乘。此时 fact(1)已知，不必再向前推了。然后开始第二阶段“回归”，从已知 fact(1)的值 1，回归出 fact(2)的值是 2，从 fact(2)的值回归出 fact(3)的值是 6。也就是说，一个递归问题可分为“递推”和“回归”两个阶段，要经过若干步才能求出最后的值。

程序执行过程中，首先 main()函数以 3 为参数调用 fact()函数，fact(3)依赖于 fact(2)的值，所以必须先计算出 fact(2)才能求 fact(3)，而要求 fact(2)又必须先求出 fact(1)。当 fact(3)递归调用自己计算 fact(2)时，fact(3)并未结束，而是暂停一下，等算出 fact(2)后再继续计算 fact(3)。当 fact(2)递归调用自己计算 fact(1)时，fact(2)也只是暂停一下并未结束，等算出 fact(1)后再继续计算 fact(2)。这样当调用 fact(1)时，计算机内部 main()、fact(3)、fact(2)、fact(1)这 4 个函数同时在运行，且都没有结束。到有了 fact(1)的确切值后，不断返回，最后计算出 fact(3)，返回到 main()函数，结束程序执行，如图 5.9 所示。

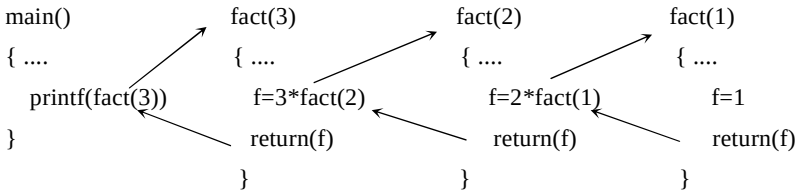


图 5.9　计算 fact(3)的调用过程

从编写递归函数程序的角度看，必须抓住两个关键点，两者缺一不可：

① 递归出口：即递归结束的条件，何时不再递归调用下去。

② 递归公式：即递归的表达式，如例 5.12 中 n!=n*(n−1)!。

求 n!也可以不用递归的方法来实现，如可以用递推法，即从 1 开始乘以 2，再乘以 3……直到 n。递推法比递归法更容易理解和实现。但是有些问题则只能用递归算法才能实现，典型的问题是 Hanoi 塔(汉诺塔)问题。

【例 5.12】Hanoi 塔问题。古代有一个梵塔，塔内有 A、B、C 三个座，A 座上放着 64 个大小不等的盘子，其中大盘在下，小盘在上。有一个老和尚想把这 64 个盘子由 A 座搬到 B 座，但一次只允许搬一个盘，且在搬动过程中，3 个座上始终都保持大盘在下，小盘在上。

在搬动过程中可以利用 C 座，要求编程输出搬动的步骤。

【分析】读者不大可能直接写出搬动盘子的每一个具体步骤，不妨先拿 3 个盘子来模拟一下，看能否找到某些重复性规律，以便用循环来实现。通过分析可以看到虽然搬动步骤的类型不多，也有重复的要求，但重复步骤不同，无明确的规律，故无法用循环实现。

现从递归的两个关键点出发，分析如何使用递归来实现搬动过程。

① 递归出口：如果只有一个盘子，可以直接搬动。

② 递归公式：如何把搬动 64 个盘子的问题简化成搬动 63 个盘子的问题？假设要搬 64 个盘子的老和尚可以命令和尚甲，把上面 63 个盘子从 A 座先搬到 C 座，然后老和尚自己把最大的盘子从 A 座搬到 B 座，再命令和尚甲把 63 个盘子从 C 座搬到 B 座，整个任务就完成了。

但还有一个问题，和尚甲又如何搬动 63 个盘子呢？如果和尚甲可以命令和尚乙搬动其中的 62 个盘子，和尚甲也可以轻松地解决问题。从编程的角度出发，我们不必关注实现的具体细节，程序给出的是解决问题的规律，而不是将各个具体步骤一一展开。因此，不必关心 63 个盘子怎么搬，62 个盘子又怎么搬……我们只需找到解决问题的规律，让计算机来实现具体的细节。

经过分析可知，将 n 个盘子从 A 座搬到 B 座可以分解为 3 个步骤：

① 把 A 座上的 n−1 个盘子搬到 C 座上；

② 把 A 座上的一个盘子(n 号盘子)搬到 B 座上；

③ 把 C 座上的 n−1 个盘子搬到 B 座上。

其中第 1 步和第 3 步是类似的，都是把 n−1 个盘子从一个座搬到另一个座，只是座的名字不同而已。按照搬动规则，必须有 3 个座才能完成搬动，一个座是搬动源，一个座是目的地，还有一个座作为中间过渡。在搬动过程中 3 个座的地位是动态变化的，因此，在函数中必须指定 3 个座，使其作为函数的参数。具体搬动步骤，用 printf()函数输出。

参考代码：

```
#include<stdio.h>
int  main(void)
{
    int  n;
    void  hanio(int  n, char  a, char  b, char  c);      //函数声明
    printf("Input  the  number  of  disks: " );
    scanf("%d", &n);
    printf("The  steps  for  %d  disks  are:\n", n);
    hanio(n, 'A', 'B', 'C');                              //函数调用
    return  0;
}
void  hanio(int  n, char  a, char  b, char  c) //定义从 a 到 b 搬动 n 个盘的函数，c 为中间过渡
{
    if(n == 1)                              //递归出口
       printf("%c−−>%c\n", a, b);
    else
```

```
    {   hanio(n-1, a, c, b);              //递归调用
        printf("%c-->%c\n", a, b);
        hanio(n-1, c, b, a);              //递归调用
    }
}
```

运行实例：

```
Input the number of disks: 3
The steps to move 3 disks:
A-->B
A-->C
B-->C
A-->B
C-->A
C-->B
A-->B
```

从运行结果可以看到，对 3 个盘子需要搬动 7 次。不难证明，对 n 个盘子需要搬动 2^n-1 次。要搬动 64 个盘子共需 $2^{64}-1$ 次，假设和尚每天 24 小时不停地搬，并且每秒钟搬动一次，大约需要 6×10^{11} 年，即约 600 亿年，比地球的年龄还要长。所以有人戏称，当老和尚搬完 64 个盘子时，“世界末日”也到了。

设计递归程序时，关键是找出运算规律，即递归公式，千万不要局限于实现细节，否则很难理出头绪，具体实现细节应该让计算机去处理。

【练习 5.8】下列程序的输出结果是_____。

```
#include<stdio.h>
long  fib(int  g)
{
    switch(g)
    {
        case  0:  return(0);
        case  1:
        case  2:  return(2);
    }
    printf("g=%d, ", g);
    return(fib(g-1) + fib(g-2) );
}
void main()
{
    long  k;
    k = fib(4);
    printf("k=%ld\n", k);
}
```

【练习 5.9】已知一个数列从第 1 项开始的前 3 项为 0、0、1，以后的各项都是其相邻的前 3 项之和。要求使用递归方法编写函数 fun(n)，求该数列第 n 项的值，返回值为长整型，并写出相应的主函数。

5.6 数组作函数参数

数组可以作为函数的参数进行数据传送。数组作函数参数有两种形式：一种是把数组元素作为实参使用；另一种是把数组名作为函数的形参和实参使用。

5.6.1 数组元素作函数实参

数组元素可以看作与之相同类型的普通变量，因此它作为函数实参与普通变量作为函数实参是完全相同的，函数形参为变量。实参和形参之间是单向传递，即“值传递”。

【例 5.13】编写程序，求 s=1!+4!+5!+7!+9!的值。

【分析】本程序是一个求累加和的问题，每一项都是一个阶乘，所以首先要求出阶乘值，然后累加。可定义 fact()函数求阶乘，而 1, 4, 5, 7, 9 这些数之间没有规律可循，可以考虑先把这些数放在一个整型数组中，然后让数组元素分别作为函数的实参调用 fact()函数。在 main()函数中用循环语句求累加和，每次累加的是函数调用 fact(a[i])的值。

参考代码：

```
#include<stdio.h>
double  fact(int  n)
{
    int  i;
    double  result = 1;
    for (i = 1; i <= n; i++)
          result = result * i ;
    return  result ;
}
int  main(void)
{    int  a[5]={1, 4, 5, 7, 9};
    int  i, n = 5;
    double  sum;
    sum = 0;
    for(i = 0; i <n; i++ )
        sum = sum + fact(a[i]);          //函数调用，数组元素做实参
    printf("sum = %e\n", sum);
    return  0;
}
```

运行实例：

```
sum=3.680650e+005
```

5.6.2　一维数组名作函数参数

数组名代表数组首元素的地址。用数组名作为实参，向形参传递数据，实际上传递的是数组的起始地址。其意义是形参可以借助数组的起始地址，实现对数组元素的读写，从而实现主调函数和被调函数操作同一个数组的效果。

用数组名作为函数实参，形参应当用数组名或指针变量，第 6 章将详细介绍指针变量作函数形参的有关内容。

【例 5.14】求任意 10 个数的平均值，结果保留两位小数。

【分析】将 10 个数存放在一个一维数组 a 中，先通过循环求 10 个数的累加和，然后用累加和除以 10 求出平均值。本程序可定义一个 ave()函数求 10 个数的平均值，用数组名作函数实参，则形参也应用数组名。形参数组不定义长度，定义 ave()函数需要两个参数，一个是数组名用来接收实参数组的首地址；另一个是整型变量，用来接收实参传过来的数组元素的个数。

参考代码：

```
#include<stdio.h>
int  main(void)
{
    int  i, a[10];
    double  ave(int  x[], int  n);                  // 函数声明
    printf("Input  10  integers: ");
    for  (i = 0 ; i < 10; i++)
        scanf("%d", &a[i]);
    printf("Average  is  %.2f\n", ave(a,10));       // 函数调用，数组名作实参
    return  0;
}
double  ave(int  x[], int  n)                       // 函数定义，形参为数组形式
{
    int  i;
    double  s=0;
    for(i = 0; i <n; i++)
      s=s+x[i];
    return  s/n;
}
```

运行实例：

```
Input  10  integers: 3  8  2  9  6  4  15  1  10  7
Average  is  6.50
```

【说明】① 用数组名作函数参数时，应该在主调函数和被调函数分别定义数组，如例 5.14 中 x 是形参数组名，a 是实参数组名，分别在其所在的函数 ave()和 main()中定义。

② 用数组名作函数参数时，实参数组与形参数组的类型必须一致。

③ 用数组名作函数参数时，传递的是数组的首地址，编译系统对形参数组大小不做检查。

因此形参数组也可以不指定大小，但需要另设一个参数，用以传递数组元素的个数。

用数组名作为函数参数时，形参数组和实参数组共占同一段内存单元。例 5.14 中 a 为实参数组，设 a 占有以 2000 为首地址的一段内存单元。x 为形参数组名。当函数调用时，实参数组 a 的首地址传给形参数组名 x，从而 x 也取得该地址 2000。于是 a 和 x 两个数组共同占有以 2000 为首地址的一段连续内存单元，如图 5.10 所示。

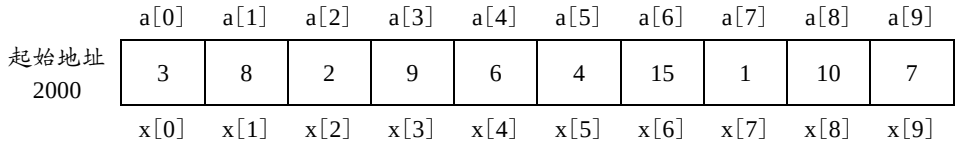


图 5.10　形参数组和实参数组共占同一内存单元

从图 5.10 可以看出，a 和 x 下标相同的元素实际上也占相同的内存单元。如 a[0]和 x[0]共占一个单元，a[0]和 x[0]相等，依次类推则有 a[i]等于 x[i]。如果改变 x[i]的值，a[i]的值也就相应地改变。也就是说，数组名作函数参数时，如果形参数组中元素的值发生变化，实参数组中元素的值也同时发生变化。

【例 5.15】输入 5 个互异的整数存入一维数组 a 中，再输入一个整数 x，然后在数组中查找 x，如果找到，输出相应的下标；否则输出“not found”。定义并调用 search(a, x)函数查找 x。

【分析】本题要求用自定义函数实现查找，因为查找的结果有两种情况，所以函数返回值也应该有两种情况，可以设定若在数组中找到 x 则返回相应的下标，若没找到则返回-1。search()函数有两个参数，第一个参数是数组名，第二个参数是整型变量。

参考代码：

```
#include<stdio.h>
#define  N  5
int  main(void)
{
    int  i, x, index;
    int  a[N];
    int search (int a[],int x);
    printf("Enter  %d  integers: ", N);
    for(i = 0; i <N; i++)
        scanf("%d", &a[i]);
    printf("Enter  x: ");
    scanf("%d", &x);
    index=search(a, x);      //函数调用，数组名作实参，在数组 a 中查找 x
    if(index== -1)
        printf("Not  Found!\n");
    else
        printf("Index  is:%d\n", index);
    return  0;
}
```

```
int search (int a[],int x)
{
    int  i;
    for(i = 0; i < N; i++)
        if(a[i] == x)                    //如果在数组 a 中找到了 x
            break;                       // 跳出循环
        if(i==N)
            return -1;                   //没找到，返回-1
        else
            return  i;                   //找到，返回下标值
}
```

运行实例 1：

```
Enter  5  integers: 2  9  8  1  6
Enter x: 9
Index  is: 1
```

运行实例 2：

```
Enter  5  integers: 2  9  8  1  6
Enter  x: 7
Not  found!
```

【例 5.16】输入一个正整数 n(n≤10)，再输入 n 个数，将它们从小到大排序后输出。定义并调用 sort(x, n)函数进行排序。

【分析】在例 4.4 中已经详细介绍过选择法排序的思路，本题要求排序的过程用自定义的 sort()函数来实现。sort()函数有两个参数，第一个参数 x 是数组名，第二个参数 n 是整型变量。在主函数中输入数据，然后调用函数，最后输出结果。使用数组名作为函数参数，在 sort()函数中改变形参数组的内容，主函数中实参数组的内容也会发生相应地变化。

参考代码：

```
#include<stdio.h>
void  sort(int  a[], int  n);                // 函数声明
int  main(void)
{
    int  i, n, a[10];
    printf("Enter  n(n<=10): ");
    scanf("%d", &n);
    printf("Enter  %d  integers: ", n);
    for(i=0; i<n; i++)
        scanf("%d", &a[i]);
    printf("Before  sorted:");               // 输出排序前的数组元素
    for(i=0; i<n; i++)
        printf("%3d", a[i]);
    printf("\n");
```

```
    sort(a, n);                          // 调用 sort()函数
    printf("After sorted:");             // 输出排序后的数组元素
    for(i=0; i<n; i++)
        printf("%3d", a[i]);
    printf("\n");
    return 0;
}
void sort(int x[], int n)                // 函数定义，用选择法进行排序
{
    int k, i, index, temp;
    for(k = 0; k < n-1; k++)
    {
        index = k;
        for(i = k + 1; i < n; i++)
        if(x[i] < x[index])
            index = i;
            temp = x[index];
            x[index] = x[k];
            x[k] = temp;
    }
}
```

运行实例：

```
Enter n(n<=10): 6
Enter 6 integers: 3 5 2 8 -1 12
Before sorted:   3  5  2  8  -1  12
After sorted:  -1  2  3  5  8  12
```

从运行结果可以看到，在执行函数调用语句“sort(a, n);”之后，a 数组已经排好序了，这是因为对形参数组 x 已用选择法进行了排序，而形参数组 x 与实参数组 a 是同一个数组。

5.6.3 二维数组名作函数参数

用二维数组名作函数参数，在函数定义时，形参数组可以指定每一维的大小，也可以省略第一维的大小说明，但不能省略第二维的大小说明。例如：“int a[3][4];”或“int a[][4];”都是合法的参数说明，而且二者是等价的；而“int a[3][];”是不合法的。

二维数组名作函数形参时的类型说明，本质上与一维数组名作函数形参时的类型说明是一致的，它们的值都代表数组的起始地址。二维数组是由若干个一维数组组成的，在内存中数组是按行存放的，因此，在定义二维数组时必须指明列数，即一行中包含几个元素。如果将形参类型说明 int a[][4]写成 int a[3][]，必然会出现语法错误，因为函数无法判断数组起始地址之后的内存单元分配。

【例 5.17】输入一个 3 行 3 列方阵，编写函数实现方阵转置。

【分析】矩阵转置，即行列互换，例 4.10 已详细介绍过实现矩阵转置的思路。本题要求编写函数实现矩阵转置，实参应该是二维数组名，形参也应该是二维数组形式。

参考代码：

```
#include<stdio.h>
int  main()
{
    int  i, j;
    int  a[3][3];
    void  trans(int  a[][3]);                    // 函数声明
    printf("输入 3 行 3 列矩阵：\n");             // 输入二维数组
    for(i=0; i<3; i++)
        for(j=0; j<3; j++)
            scanf("%d", &a[i][j]);
    trans(a);                                    // 函数调用，数组名做实参
    printf("转置后矩阵：\n");                     // 按矩阵形式输出二维数组
    for(i=0; i<3; i++)
    {
        for(j=0;j<3;j++)
            printf("%d ", a[i][j]);
        printf("\n");
    }
    return  0;
}
void  trans(int  a[][3])                         //函数定义，形参是二维数组
{
    int i, j, k;
    for(i=0; i<3; i++)
        for(j=i+1; j<3; j++)
        {
            k=a[i][j];
            a[i][j]=a[j][i];
            a[j][i]=k;
        }
}
```

运行实例：

```
输入3行3列矩阵：
1  2  3
4  5  6
7  8  9
转置后矩阵：
```

```
1  4  7
2  5  8
3  6  9
```

【练习 5.10】输入一个正整数 n(n≤10)，编写 fun(a, n)函数，其功能是计算 n 门课程的平均分，计算结果作为函数的返回值。正整数 n 和 n 门课程的成绩在主函数中输入。

【练习5.11】有以下程序：

```
#include<stdio.h>
int f(int a)
{
    return a%2;
}
void main()
{
    int s[8]={1, 3, 5, 2, 4, 6}, i, d=0;
    for(i=0; f(s[i]); i++)    d+=s[i];
    printf("%d", d);
}
```

运行程序后的输出结果是_____。

A. 9　　B. 11　　C. 19　　D. 21

【练习5.12】有以下程序：

```
#include<stdio.h>
void fun(int a[], int n)
{
    int i, t;
    for(i=0; i<n/2; i++)
    {t=a[i];  a[i]=a[n-1-i];  a[n-1-i]=t;}
}
void main()
{
    int k[10]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10},i;
    fun(k, 5);
    for(i=2; i<8; i++)    printf("%d", k[i]);
    printf("\n");
}
```

运行程序后的输出结果是_____。

A. 345678　　B. 876543　　C. 1098765　　D. 321678

【练习5.13】有以下程序：

```
#include<stdio.h>
int f(int b[][4])
{
    int i, j, s=0;
```

```
    for(j=0; j<4; j++)
    {   i=j;
        if(i>2)   i=3-j;
        s+=b[i][j];
    }
    return s;
}
void main()
{
    int a[4][4]={{1, 2, 3, 4},{0, 2, 4, 6},{3, 6, 9, 12},{3, 2, 1, 0}};
    printf("%d", f(a));
}
```

运行程序后的输出结果是_____。

A. 12　　B. 11　　C. 18　　D. 16

习　题　5

1. 输入两个正整数 m 和 n(m>n)，编写程序求 $C_m^n=\dfrac{m!}{n!(m-n)!}$ 的值。要求定义和调用 fact(n)函数计算 n!。

2. 输入精度 e，使用格里高利公式求 π 的近似值，精确到最后一项的绝对值小于 e。要求定义和调用 funpi(e)函数求 π 的近似值。格里高利公式为：$\dfrac{\pi}{4}=1-\dfrac{1}{3}+\dfrac{1}{5}-\dfrac{1}{7}+$

第6章 指　　针

指针是C语言的一个重要部分,是最能体现C语言优点的部分,也是初学者不易掌握的内容之一。使用指针可以更好的完成任务,有些工作不使用指针将无法实现。要学好 C 语言,指针是必须掌握的内容。

知识结构

1. 指针的概念
2. 指针与简单变量
3. 指针与数组
   ① 指针的运算
   ② 数组名作函数参数
4. 指针与字符串
   ① 使用指针表示字符串
   ② 动态内存分配
   ③ 常用字符串处理函数
5. 指针进阶
   ① 二级指针
   ② 指针与二维数组
   ③ 指针数组
   ④ 命令行参数
   ⑤ 返回指针的函数
   ⑥ 函数指针

6.1 指针的概念

计算机的发展速度日新月异，但是直到目前为止，计算机还沿用冯·诺依曼的体系结构，这样的计算机称为冯·诺依曼结构计算机。冯·诺依曼设计思想主要有以下三点：

① 计算机硬件的组成包括运算器、存储器、控制器、输入设备和输出设备五大基本部件。

② 计算机内部采用二进制。

③ 程序和数据存储在计算机的内存中，能自动执行，无需人干预。

其核心是存储程序的思想，即程序和数据存储在计算机的存储器中，自动执行。程序和数据都要占据一定的存储空间，计算机的存储器分内存储器(也称主存储器)、外存储器(也称辅存储器)。CPU 能直接访问的是内存，也就是程序和数据需要存入到内存中才能被执行。计算机内存基本单位是字节(byte)，每 8 位二进制位(bit)组成的字节称为一个内存单元，整个计算机的内存就是由若干内存单元组成的，为了区分各个内存单元，计算机对每个内存单元定义了一个唯一的编号，称为内存单元的地址，就好比是房间号，起始从 0 开始，按照字节的顺序编址，内存单元里存储的程序和数据即存储的内容，就好比是房间里住的房客。对内存单元的操作只有两种：写操作和读操作，写操作就是把数据存入内存单元，读操作就是取出内存单元中的数据。无论读或者写，都是通过该内存单元的地址进行的，因此，知道了某内存单元的地址就可以对这个内存单元进行读写。

C 语言中把内存地址作为指针来处理，一个变量在内存中的地址称为这个变量的指针，指针是 C 语言中的一种特殊的数据，指示变量、函数、数据在内存中的存放位置，也就是某内存单元的地址，简单说，指针就是地址。

C 语言中的变量要先定义，后使用。例如：

```
char  c;
int  i, j;
```

编译系统在对以上变量定义进行编译时，就会在内存中为这些变量分配存储单元，目前常用的系统中，整型变量占 4 个字节，字符型变量占 1 个字节。数据占据连续几个字节的内存空间，叫做存储块，首地址是指这个块的第一个字节的地址，这个地址就是变量的地址，也称为变量的指针。

系统给上面定义的变量分配内存，如图 6.1 所示，变量 C 占据地址为 1044964 的这个字节。变量 i 占据从地址 1044960 开始的连续四个字节的存储块，变量 j 占据从地址 1044956 开始的连续四个字节的存储块，1044964，1044960，1044956 是存储块的首地址，也就是变量 c，i，j 的地址，也是变量 c，i，j 的指针。

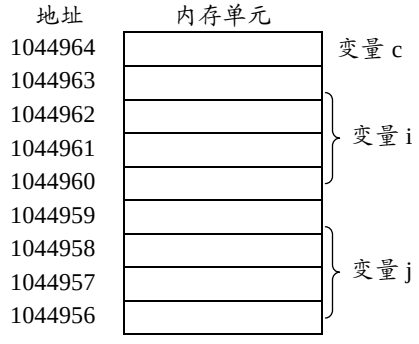


图 6.1　内存中的变量的地址

在定义了这些变量之后，程序中就可以使用这些变量了，例如：

```
i=1;          c='A';          printf("%d", i);
```

为了正确地访问这些变量所在的内存单元，编译系统需要在变量名与变量的地址之间建立关联，这样就可以直接按照变量名对这些变量对应的单元进行读写，这种方式称为对内存的直接访问。除了直接访问方式，还有间接访问方式，就是把一个变量的地址存放到另外一个专门存放地址的变量中，先访问这个存放地址的变量，得到这个变量的地址后，再根据这个地址访问到该变量，这就是间接访问方式。

C 语言中定义这种专门存放地址的变量为指针变量。指针变量的内容是指针即地址。例如：

```
int  i ,*i_point;              // 定义一个整型变量 i，一个整型指针变量 i_point
i_point=&i;                    // 将变量 i 的地址赋值给指针变量 i_point
```

i_point是一个指针变量，也要占据内存单元，假如分配给它的内存单元的地址是1044924，

那么，它的内容是变量 i 的地址(本例中是 1044960)，即变量 i 的指针。可以说指针变量 i_point 指向了变量 i，i_point 是一个指向 i 的指针，如图 6.2 所示。

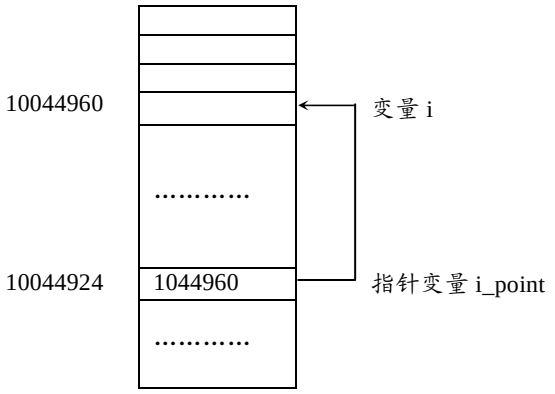


图 6.2　指针变量与指针

通常在不引起混淆的情况下，指针变量也简称为指针，这时应根据上下文正确理解。

指针不但可以指示变量在内存中的位置，也可以指示数据和函数或者指针变量在内存中的位置。函数在内存中的入口地址称为函数的指针；使用动态内存分配函数得到的内存块，其首地址称为它的指针，这个内存块没有名称，只能使用指针来访问。如果一个指针变量的地址再存到一个指针变量里，即存放指针变量的指针，就是二级指针，同样的可以有多级指针。这些内容将在后面介绍。

【总结】① 指针就是地址。

② 指针变量就是存储指针的变量。

③ 变量的指针是变量所占存储空间的首地址。

6.2　指针与简单变量

6.2.1　指针变量的定义与引用

1. 定义指针变量

和其他变量一样，使用指针变量之前必须定义。指针变量的命名规则与其他变量一样，遵循 C 语言自定义标识符的命名规则。

定义指针变量的格式如下：

基类型 *指针变量名;

由此，一个指针变量有三个要素：变量名，指针，类型。首先它是一个变量(变量名)，其次用“*”符号表示它是一个指针变量，这个指针变量只能指向特定的数据类型(基类型)的数据。例如：

```
int  *i_point;          // 定义一个指向整型数据的指针变量 i_point
char  *c_point;         // 定义一个指向字符型数据的指针变量 c_point
double  *d_point;       // 定义一个指向双精度浮点型数据的指针变量 d_point
```

【注意】“*”在变量的定义中出现，只表明定义的是指针变量，它不是变量的一部分。编译系统也能够根据上下文判断“*”是乘法运算符还是间接运算符或者是指针变量定义符号。

2. 初始化指针

定义了指针变量后，必须对其进行初始化才能使用。使用没有初始化的指针结果是不可预测的。将地址存储到指针中，或者说将地址赋值给指针，就是指针变量的初始化。

例如：

```
int  a;
int  *a_point=&a;          // 定义一个指向整型变量的指针 a_point
                           // 并将整型变量 a 的地址赋值给 a_point
int  a, *a_point;          // 定义一个整型变量 a 和一个指向整型变量的指针 a_point
a_point=&a;                // 将整型变量 a 的地址赋值给 a_point，
                           // 即对 a_point 进行初始化，a_point 指向 a
```

【注意】只能将地址赋值给指针，不能将一个整数或者其他数值赋值给指针。

例如执行语句“a_point=2000;”会造成严重的错误。

NULL 表示空指针，NULL 是一个符号常量，值为 0。当使用 NULL 初始化指针时，称这个指针为空指针，表示该指针不指向任何对象。例如：

```
a_point= NULL;          // 对指针进行初始化，将空指针赋值给指针变量。
```

3. 使用指针

定义指针并初始化后，就可以引用指针变量了，对指针变量常使用以下两个运算符：

① 取地址运算符&：&加在变量名的前面，就是取变量的地址，&a 就是变量 a 的地址。例如：

```
int  a, b;
scanf("%d%d", &a, &b);              //调用 scanf()函数通过键盘给变量 a，b 赋值
```

&运算符常用来给指针变量赋值，例如：

```
int  a, b, *a_point, *b_point;
a_point=&a;   b_point=&b;           //给指针变量 a_point，b_point 赋值
scanf("%d%d", a_point, b_point);    //调用 scanf()函数，通过键盘给变量 a，b 赋值
```

② 间接运算符*：也称指针运算符，加在指针变量的前面，间接访问指针变量指向的内存单元，即读取指针变量所指向的内存单元的值或者向指针变量指向的内存单元写入值。

编译系统会根据上下文判断“*”的含义。例如：

```
int  a=5, b, * a_point=&a;          // 定义指针变量并赋初值，*用来定义指针变量
b=*a_point+10;                      // *为间接运算符
                                    // 取指针变量 a_point 指向单元的值，即变量 a 的值
*a_point=b*5;                       // 将变量 b 的值乘以 5 后
                                    // 赋值给指针变量 a_point 指向的单元，即变量 a。
```

【例 6.1】指针变量的定义和引用示例。

```
#include <stdio.h>
int  main()
{
    int  a=10, b=5;             // 定义 2 个整型变量 a，b，并初始化
    int  *pa, *pb;              // 定义 2 个指向整型变量的指针变量 pa，pb

    pa=&a;                      // 给指针变量 pa 赋值，使它指向变量 a
    pb=&b;                      // 给指针变量 pb 赋值，使它指向变量 b

    printf("%d, %d\n", a, b);   // 输出变量 a，b 的值，直接访问
    printf("%d, %d\n", *pa, *pb);   // 输出变量 a，b 的值，间接访问
```

```
    printf("%d, %d\n", &a, &b);          // 输出变量 a，b 的地址
    printf("%d, %d\n", pa, pb);          // 输出变量 a，b 的地址

    return 0;
}
```

运行实例(不同系统中输出的地址信息不尽相同)：

```
10, 5
10, 5
1245052, 1245048
1245052, 1245048
```

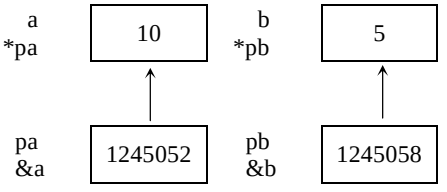


图 6.3　指针变量的引用

如图 6.3 所示，变量 a 的地址赋值给指针 pa，即指针 pa 指向变量 a，于是访问变量 a 可以直接访问，也可以间接访问，即 a 与*pa 指的是变量 a 的内容，变量 a 的地址&a 就是指针 pa，&a 与 pa 指的是变量 a 的地址。

【例 6.2】指针变量赋值示例，如图 6.4 所示。

```
#include<stdio.h>
int  main()
{
    double  a=0, b=6.0, *pa, *pb;
    pa=&a;
    scanf("%lf", pa);      // pa 即&a
    *pa=*pa+2;             // 对指针变量 pa 间接运算，*pa 即 a，该语句等价于 a=a+2
    pb=pa;                 // pa 与 pb 是相同类型的指针，可以互相赋值，结果相当 pb=&a
    pa=&b;                 // 给 pa 再次赋值，使之指向 b
    printf("%lf, %lf", *pb, *pa);    // 对指针变量间接运算
                                     // *pb 间接访问变量 a，*pa 间接访问变量 b
    return 0;
}
```

运行实例：

```
1.0
3.000000, 6.000000
```

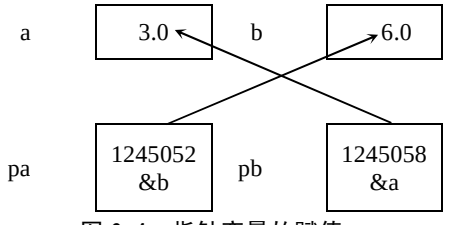


图 6.4　指针变量的赋值

6.2.2　指针变量的特殊性

指针变量作为 C 语言中一种特殊变量类型，使用非常灵活，能够完成一些特殊的操作，但是如果使用不当会引起严重的后果。

作为变量的一种，指针变量也遵循先定义后使用，先赋值后引用的原则。同时，定义了一个指针变量后，编译系统也要为这个指针变量分配一定大小的存储空间。所分配的存储空间的大小只和系统有关，而和指针的基类型无关，也就是说，无论是指向什么类型的指针，在特定的系统中所分配的内存空间的大小是固定的。

指针变量有以下的特殊性：

① 指针变量中只能存放内存单元的地址，而不能存放一般意义的数据；

② 指针变量必须初始化后才能引用，使用没有指向的指针变量可能造成严重的错误；

③ 基类型不是指针变量的类型，而是指针所指向的对象的类型。所以定义指针时要说明该指针指向的对象的类型，使用时不能将不同类型的变量的地址赋值给同一指针；

④ 有些运算对指针变量是没有意义的，如乘法、除法等，指针变量能够参与的运算在后面详细介绍。

【注意】不同数据类型的变量占用的内存空间大小随系统的不同而不同，不是固定的。目前常用系统中 short 型变量占 2 个字节，int 型变量占 4 个字节，double 型变量占 8 个字节，char 型变量无论在什么系统中都是 1 个字节。

指针是变量所占用内存块的第 1 个字节的地址(低地址)，即变量的首地址。

【例 6.3】非法使用指针示例。

```
#include<stdio.h>
int  main()
{
    int  *p, a=1;
    double  *q;
    *p=5;                    //错误！指针 p 未初始化，所以*p 是无意义的
    q=&a;                    //错误！指针 q 只能指向 double 型的变量
    printf("%x, %d, %d, %f", p, *p, *q, *q);
    return  0;
}
```

编译过程给出发生错误的信息，运行后不能得到正确结果。

例 6.3 中“q=&a;”语句出错是因为 q 定义为指向 double 型变量的指针，而 a 为 int 型变量。如果在“*p=5;”语句前面加上“p=&a;”语句，指针 p 就初始化为指向 a 的指针。a 占 4 个字节，那么 p 就是这 4 个字节中的第一个字节的地址。“*p=5;”就是合法的语句了。

6.2.3　指针变量作为函数的参数

指针变量同样可以作为函数的参数，传递的是地址数据。使用时要注意形参的类型，实参必须是已经赋值的地址。

比较以下四个程序的运行效果。

【例 6.4】函数的形参是普通整型变量。

```
#include<stdio.h>
void  main()
{
    void  swap1(int  x, int  y);
    int  a=15, b=8;
    swap1(a, b);
    printf("a=%d, b=%d", a, b);
}
```

```
void swap1(int x, int y)
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```

运行实例：

```
a=15, b=8
```

调用函数时，实参 a 的值传递给形参 x，b 的值传递给形参 y，如图 6.5 的左图所示。执行函数 swap1 的代码，形参 x 和 y 的值交换了，返回主函数形参 x 和 y 就释放了，并未影响实参 a 和 b 的值。如图 6.5 的右图所示。

【例 6.5】函数的形参是指针变量示例。

```
#include<stdio.h>
void main()
{
    void swap2(int *x, int *y);          //函数声明，形参是指针
    int a=15, b=8;
    swap2(&a, &b);                       //函数调用，实参是地址
    printf("a=%d, b=%d", a, b);
}
void swap2(int *x, int *y)               //函数首部，形参是指针
{
    int temp;                            //定义整形变量 temp
    temp = *x;                           //对形参变量做间接运算，实际访问的是实参变量
    *x = *y;
    *y = temp;
}
```

调用过程时，实参&a 的值传递给形参 x，&b 的值传递给形参 y，结果是形参指针 x 和指针 y 指向变量 a，b，如图 6.6 的左图所示。执行函数 swap2 的代码，*x 和*y 的值交换了，通过对指针变量的间接运算，使得变量 a 和 b 的值进行了交换，如图 6.6 的右图所示。

程序输出结果为 a=8, b=15。

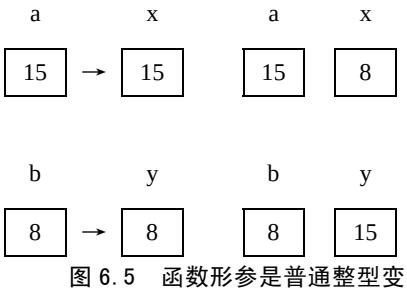


图 6.5　函数形参是普通整型变量

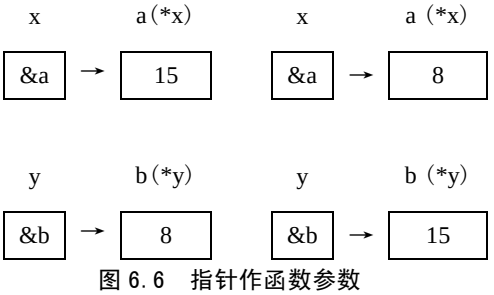


图 6.6　指针作函数参数

【例 6.6】将例 6.5 中的 swap2()函数改为以下的 swap3()函数，考查结果。

```
void  swap3(int  *x, int  *y);        //函数首部，形参是指针
{
    int  *temp;                       //定义指针变量 temp
    temp = x;                         //对指针进行赋值操作，没有进行间接操作
    x = y;
    y = temp;
}
```

用语句“swap3(&a, &b);”调用函数，则实参&a 的值传递给形参 x，&b 的值传递给形参 y，结果是形参指针 x 和指针 y 指向变量 a，b。执行函数 swap3()的代码后，形参 x 和 y 的值进行了交换，x 指向 b，y 指向 a，如图 6.7 所示，变量 a 和 b 的值并未改变。

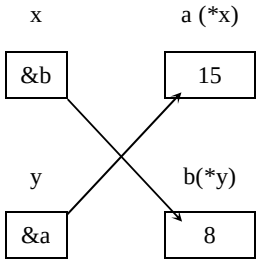


图 6.7　调用 swpa3 函数结果

【例 6.7】将例 6.5 中的 swap2 函数改为以下的 swap4 函数，考查结果。

```
void  swap4(int  *x, int  *y);    //函数首部，形参是指针
{
    int  *temp;             //定义指针变量 temp
    *temp = *x;             //指针 temp 未初始化，对其间接访问是错误的
    *x = *y;
    *y =*temp;
}
```

*temp 是访问指针变量 temp 所指向的存储单元，但是 temp 并未赋值，它的指向是不确定的，所以对*temp 赋值是一危险的操作，是错误的。

C 语言中，函数参数传递为单向值传递，实参是指针，传递的是变量的地址，通过间接操作，就能实现将函数内的操作结果带回到主函数。

【练习 6.1】① 有以下程序：

```
void  main()
{   int  a=7, b=8, *p, *q, *r;
    p=&a;   q=&b;
    r=q;    q=p;   p=r;
    printf("%d, %d, %d, %d", *p, *q, a, b);
}
```

运行程序后的输出结果是_____。

A. 8, 7, 8, 7　　B. 7, 8, 7, 8　　C. 8, 7, 7, 8　　D. 7, 8, 8, 7

② 若有“int n=7, *p=&n, *q=p;”，以下语句中非法的赋值语句是_____。

A. p=q;　　B. *p=*q;　　C. n=*q;　　D. p=n;

③ 若有“int i=0, j=1, *p=&i, *q=&j;” 以下语句中非法的赋值语句是_____。

A. i=*&j;　　B. p=&*&i;　　C. j=*p;　　D. i=*&q;

④ 有以下程序：

```
void  fun(char  *a, char  *b)
```

```
{   a=b;   (*a)++;}
void  main()
{   char  c1='A', c2='a', *p1, *p2;
    p1=&c1;
    p2=&c2;
    fun(p1, p2);
    printf("%c%c", c1, c2);
}
```

运行程序后的输出结果是_____。

A. Ab　　B. aa　　C. Aa　　D. Bb

【练习6.2】有以下程序，运行程序后的输出结果是_____。

```
void  swap(int  x, int  y)
{
    int  t;
    t=x;    x=y;    y=t;
    printf("%d  %d", x, y);
}
int  main()
{
    int  a=3, b=4;
    swap(a,b);
    printf("%d %d", a, b);
    return  0;
}
```

【练习 6.3】以下程序运行后的输出结果是_____。

```
void  swap(int  *x, int  *y)
{
    int  t;
    t= *x;    *x = *y;    *y =t;
    printf("%d  %d ", *x, *y);
}
int  main()
{
    int  a=3, b=4;
    swap(&a, &b);
    printf("%d  %d", a, b);
    return  0;
}
```

【练习 6.4】以下程序运行后的输出结果是_____。

```
void  byvalue(int  x, int  y, int  z)
{
```

```
    x=0;
    y=0;
    z=0;
}
void  byref(int  *x, int  *y, int  *z)
{
    *x=0;
    *y=0;
    *z=0;
}
int  main()
{
    int  a=3, b=4, c=5;
    printf("\n  Befor  calling  byvalue(),");
    printf("a=%d, b= %d, c=%d", a, b, c);
    byvalue(a, b, c);                          //函数调用
    printf("\n  After  calling  byvalue(),");
    printf("a=%d, b= %d, c=%d ", a, b, c);
    byref(&a, &b, &c);                         //函数调用
    printf("\n  After  calling  byref(),");
    printf("a=%d, b= %d, c=%d ", a, b, c);
    return  0;
}
```

6.3　指针与一维数组

在 C 语言中，常使用指针来操作数组，这是因为指针和数组之间有着特殊的关系。数组名表示数组的起始地址，即数组名是一个指针，而且是一个指针常量。前面我们引用数组元素时使用的是下标法，实际上使用指针(即指针法)也可以方便地引用数组元素，使用指针法访问数组一般要比使用下标法访问效率高。

6.3.1　数组名是--个指针常量

前面已经明确说明数组在内存中是顺序存放的，一个数组的数组元素占用连续的一块内存单元。数组名就是这块连续内存单元的首地址。一个数组由各个数组元素(下标变量)组成。每个数组元素依其类型占有几个连续的内存单元。每个数组元素的首地址也是指它所占有的存储块的首地址。数组名是一个指向该数组的指针，它是一个指针常量，不能被修改，因为系统一旦给这个数组分配了存储区，在整个程序运行过程中是不会改变的。可以定义一个指针变量，指针变量既可以指向一个数组，也可以指向一个数组元素，可把数组名或某个元素

的地址赋予它。例如：

```
int a[100], *pa;
pa=a;
pa=&a[8];
```

这里，不带方括号的数组名 a 是一个指针常量，代表数组的首地址，也是数组第一个元素 a[0]的首地址，a 和&a[0]等价。

pa 是一个指针变量，可以通过“pa=a;”语句把数组 a 的首地址赋给指针变量 pa，即让指针 pa 指向数组 a，也可以再通过赋值语句改变它的值，使之指向某一个数组元素，如“pa=&a[8];”语句使指针 pa 指向数组元素 a[8]。数组名 a 是指针常量，不能被改变，指针 pa 是变量，可以指向其他的变量。

定义时可以给指针变量赋初值：

```
int a[100], *pa=a;          或    int a[100], *pa=&a[0];
```

等价于：

```
int a[100],*pa;
pa=a; 或 pa=&a[0];
```

C 语言中系统这样处理，如果指针变量 p 已经指向数组中的一个元素，则 p+1 指向同一数组的下一个元素(在数组元素个数范围内)，而不是将指针 p 的值(地址)简单加 1.

如果 pa=a，则数组 a 的首地址可以有几种表示法：a，&a[0]，pa，而 pa+i 和 a+i 就是 a[i]的地址，它们都指向数组 a 的第 i 个元素。如 pa+8 和 a+8 的值是&a[8]，都指向 a[8]。

数组元素 a[0]，a[1]，……，a[i]的地址也有下述几种表示法：

```
&a[0]，  &a[1]，  &a[2]，  ……，  &a[i]
a，      a+1，    a+2，    ……，  a+i
pa，     pa+1，   pa+2     ……，  pa+i
```

可以使用间接运算符“*”，*(a+i)或*(pa+i)是 a+i 或 pa+i 所指向的数组元素 a[i]。例如*(pa+8)和*(a+8)就是 a[8]。指向数组的指针变量也可以带下标，pa[i]与*(pa+i)等价。

数组元素同样可以有下述几种表示法：

```
a[0]，   a[1]，     a[2]，   ……，  a[i]
*a，     *(a+1)，   *(a+2)，  ……，  *(a+i)
*pa，    *(pa+1)，  *(pa+2)，……，  *(pa+i)
pa[0]，  pa[1]，    pa[2]，  ……，  pa[i]
```

以上表示建立在 pa=a 的前提下。以下两个例题说明了数组元素的不同表示方法。

【例 6.8】数组元素表示示例 1。

```
#include<stdio.h>
int main()
{
    int *pa, i;
    int a[10]={0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    for(pa=a, i=0; i<10; i++)
        printf("%d,%d,%d,%d,", a[i], *(a+i), *(pa+i), pa[i]);
    return 0;
}
```

运行实例：

```
0,0,0,0,1,1,1,1,2,2,2,2,3,3,3,3,4,4,4,4,5,5,5,5,6,6,6,6,7,7,7,7,8,8,8,8,9,9,9,9
```

【例 6.9】数组元素表示示例 2。

```
#include<stdio.h>
int  main()
{
    int  i, *pa, a[5]={0, 1, 2, 3, 4};
    double  b[5]={0.0, 1.0, 2.0, 3.0, 4.0}, *pb;
    pa=a;                    //相当于 pa=&a[0]
    pb=&b[4];                //相当于 pb=b+4
    *pa=5 ;
    *pb=6.0 ;
    for(i=0; i<5; i++)
    printf("%d, %4.1f,", *(pa+i), b[i]);
    return  0;
}
```

运行实例：

```
5, 0.0, 1, 1.0, 2, 2.0, 3, 3.0, 4, 6.0
```

以上实例说明数组下标表示法与数组指针表示法之间是等价的，可以使用其中任何一种。

6.3.2　指针的运算

指向数组的指针在指向数组的某个元素后，要访问其后的数组元素，必须给指针加上一个值，如要访问其前的数组元素，必须给指针减掉一个值。这些操作可以使用指针的算术运算来实现，如指针递增、递减。

指针递增是给指针加上一个值，如果 pa 是指向 int 型数组 a[10]的指针，且 pa=a，pa 指向第一个数组元素 a[0]，那么，执行 pa++后，pa 指向数组下一个元素，即 a[1]，编译器会根据数组的数据类型的长度来增加指针的值。如执行“pa=pa+5;”语句后，pa 增加的实际字节数的值是 5*4(每个 int 型变量占 4 个字节)。

指针递减是给指针加上一个负值，如 pa--，指向数组其前一个元素。

利用指针的算术运算，程序可以高效的访问数组中的元素。

【例 6.10】指针的算术运算。

```
#include<stdio.h>
int  main()
{
    int  *pa, a[10]={0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    for(pa=a; pa<(a+10); pa++)
        printf("%d    ", *pa);
    return  0;
}
```

运行实例：

```
0   1   2   3   4   5   6   7   8   9
```

注意例 6.10 中的 pa 是指针变量，所以可以通过算术运算改变其值，如果把 pa++改为 a++就错了，a 是数组名，它是指针常量，不可以进行运算改变其值。

例 6.10 中的程序也可以写成：

```
#include<stdio.h>
int  main()
{
    int  *pa, i, a[10]={0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    for(pa=a, i=0; i<10; i++)
        printf("%d     ", *pa++);
    return 0;
}
```

【注意】① 对于*pa++，由于++和*优先级相同，是自右向左结合，因此等价于*(pa++)，pa++运算中先取 pa 的值为表达式的值，所以是先对 pa 的原值进行间接运算*pa，得到现在 pa 所指的数组元素的值，然后使 pa 的值增 1，这样 pa 就指向下一个数组元素了。对 pa 赋值的语句“pa=a;”也可写为“pa=&a[0];”，则*pa 的结果是 a[0]，即为 0，pa 增 1，指向 a[1]，即 pa==&a[1]。

② 不能把*pa++写成(*pa)++，(*pa)++相当于 a[i]++，使 a[0]的值增 1，变成 1，而指针并未变化。

③ *(pa++)与*(++pa)不同，前者是先取*pa 值后使 pa 加 1。后者是先使 pa 加 1，再取*pa。若 pa 的值为&a[0]，输出*(pa++)是 a[0]的值，输出*(++pa)是 a[1]的值。

④ 若指针 pa=&a[i]，则*(pa--)是先取 pa 作*运算，再使 pa 减 1。相当于 a[i--]。而*(--pa)是先使 pa 减 1，再作*运算，相当于 a[--i]。

将++和--运算符用于指向数组的指针变量可以使指针变量自动向前或向后移动，指向前一个或下一个数组元素。可以使用这样的循环语句输出 a 数组的 100 个元素：

```
pa=a;                                  或          pa=a;
while(pa<a+100)                                    while(pa<a+100)
    printf("%d, ",*pa++);                              { printf("%d, ", *pa);
                                                         pa++;}
```

指针除了可以进行加或减一个整数的算术运算，如果有两个指向同一数组的指针，还可以进行比较和相减运算。

如 pa1，pa2 指向同一个数组的不同数组元素，则 pa1-pa2 得到的是这两个指针的距离，即数组元素的个数。

```
pa1=&a[8];  pa2=&a[6];  n=pa1-pa2;
```

n 的值是 2，表示这两个数组元素的距离，同样是元素的个数而不是字节数。

pa1 与 pa2 可以用<、<=、>、>=、==、!= 六种关系运算符进行比较，前面的数组元素(下标较小)的地址小于后面的数组元素(下标较大)的地址。

除了以上所述的运算，其他运算对指针变量没有意义，如乘法、除法等，对指针使用这些运算将导致错误。

表 6.1 列出了对指针可以进行的所有运算。

表 6.1　指针的运算

运　算	含　　义
赋值	只能用地址数据给指针变量赋值，如用&获得的地址、数组名或赋空指针 NULL
间接运算	间接运算符*，获取存储在指针指向的位置的值(解除引用)
求地址	可以使用地址运算符&获得指针的地址
递增	给指针加上一个整数，使之指向另一个内存单元，通常用在指向数组元素的指针，加上整数后不能越界
递减	给指针减去一个整数，使之指向另一个内存单元，通常用在指向数组元素的指针，减去整数后不能越界
求差	两个指针相减，得到二者之间的距离
比较	只能对指向同一个数组的指针进行比较

【练习 6.5】① 若有定义语句：

```
double x[5]={1.0, 2.0, 3.0, 4.0, 5.0}, *p=x;
```

则下述各项中，错误引用 x 数组元素的是_____。

A. *p　　B. x[5]　　C. *(p+1)　　D. *x

② 有以下程序：

```
void main()
{
    int a[10]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, *p=&a[3], *q=p+2;
    printf("%d\n", *p+*q);
}
```

运行程序后的输出结果是_____。

A. 16　　B. 10　　C. 8　　D. 6

③ 若有以下定义及语句：

```
int a[10]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, *p=&a[3], b;
b=p[5];
```

b 中的值是_____。

A. 5　　B. 6　　C. 8　　D. 9

④ 已有定义“int i, a[10], *p;”则下述语句中合法的赋值语句是_____。

A. p=100;　　B. p=a[5];　　C. p=a[2]+2;　　D. p=a+2;

6.3.3　将数组地址传递给函数

前面已经讨论了，数组名是一个指针常量，是数组的首地址，用数组名作为函数的参数，可以将数组地址传递给函数，函数中的形参是可以接受地址的指针变量，在获得了数组的地址后就可以使用指针表示法对数组元素进行操作。

① 数组名是指针常量，相当于指针作为函数的参数。

② 数组名作为实参，形参是指针变量(数组)。

【例 6.11】用数组名做函数的参数应用：计算数组元素累加和。

```
int main(void)
{
   int sum(int *array, int n);
```

```
    int b[5] = {1, 4, 5, 7, 9};
    printf("%d\n", sum(b, 5));      //函数调用，数组名做实参
    return 0;
}
int sum(int *array, int n)          // 第一个形参是指针变量
{
    int i, s = 0;
    for(i=0; i<n; i++)
        s += array[i];
    return(s);
}
```

运行实例：

```
26
```

在第 5 章中，我们已经学过用数组名作函数的参数，实质上就是传递数组的地址，所以像下面这样，把形参写成数组名的形式的效果是一样的。

```
int sum(int array[], int n)         // 第一个形参是数组名
{
    int i, s = 0;
    for(i=0; i<n; i++)
    s += array[i];
    return(s);
}
```

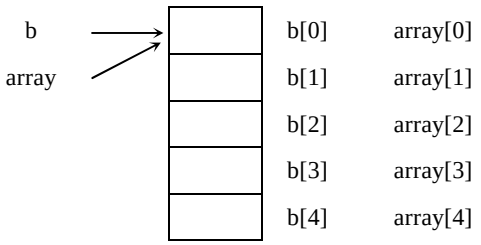


图 6.8　数组名作函数参数

主函数中，通过“sum(b, 5)”调用函数传递参数后，结果如图 6.8 所示，所以返回值 s 是计算 b[0]+b[1]+b[2]+b[3]+b[4]的结果。

用不同的参数调用函数，所对应的 s 的结果如下：

sum(b, 3)	b[0]+b[1]+b[2]
sum(b+1, 3)	b[1]+b[2]+b[3]
sum(&b[2], 3)	b[2]+b[3]+b[4]

数组名作为函数的参数，在函数调用时，将实参数组首元素的地址传给形参(指针变量形式或数组名形式)，因此，形参数组和实参数组是同一段存储空间，在函数中通过形参存取的数组元素实质上就是实参数组中的元素。

【例 6.12】逆序存放数组元素。

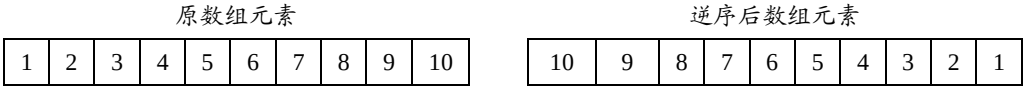


图 6.9　数组元素逆序

```
#include<stdio.h>
int main(void)
{
    int a[10]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, *p=a;
```

```
    int  t, *q;
    q=a+9;                  //指针 q 指向数组最后一个元素
    while(p<q)
    {
        t=*p;    *p = *q;    *q = t;
        p++; q--;
    }
    for (p=a; p<a+10; p++)
        printf("%d, ", *p);
    return  0;
}
```

运行实例：

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
```

【例 6.13】输入 n 个整数并放在数组中，通过调用函数的方法实现数组元素逆序存放。

```
#include<stdio.h>
int  main(void)
{
    int  i, a[10], n;
    void  reverse(int  p[], int  n);              //函数声明
    printf("Enter  n: ");
    scanf("%d", &n);
    printf("Enter  %d  integers: ", n);
    for(i = 0; i < n; i++)
    scanf("%d", &a[i]);
    reverse(a, n);                                //函数调用，数组名作实参
    for(i = 0; i < n; i++)
        printf("%d\t", a[i]);
    return  0;
}
void  reverse(int  *p, int  n)              //函数定义
{
    int  *pj, t;
    pj=p+n-1;                                 //对指针变量 pj 赋值，使之指向数组最后一个元素
    while(p<pj)
    {
        t=*p;     *p=*pj;     *pj=t;
        p++; pj--;
    }
}
```

运行实例：

```
Enter n:7
```

```
Enter 7 integers:1 2 3 4 5 6 7
7    6    5    4    3    2    1
```

【注意】用数组名作函数的参数时，通常还需要用一个参数传递数组元素的个数，如上例中的参数 n。

【练习 6.6】① 有以下程序：

```
void prt(int *m, int n)
{
    int i;
    for(i=0; i<n; i++)      m[i]++;
}
void main()
{
    int a[]={1, 2, 3, 4, 5},i;
    prt(a, 5);
    for(i=0; i<5; i++)      printf("%d, ", a[i]);
}
```

运行程序后的输出结果是_____。

A. 1, 2, 3, 4, 5,　　B. 2, 3, 4, 5, 6,　　C. 3, 4, 5, 6, 7,　　D. 2, 3, 4, 5, 1,

② 有以下程序：

```
void sum(int *a)
{   a[0]=a[1];}
main()
{   int aa[10]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, i;
    for(i=2; i>=0; i--)     sum(&aa[i]);
    printf("%d", aa[0]);
}
```

运行程序后的输出结果是_____。

A. 4　　B. 3　　C. 2　　D. 1

③ 有以下程序：

```
#include<stdio.h>
void main()
{
    int a[]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,};
    int *p=a+5,*q=NULL;
    *q=*(p+5);
    printf("%d  %d \n",*p,*q);
}
```

运行程序后的结果是_____。

A. 报错　　B. 输出6　6　　C. 输出6　12　　D. 输出5　5

【练习 6.7】下述程序的运行结果是_____。

```
#include<stdio.h>
```

```
void  f(int  b[], int  n)
{
    int  i, r;
    r=1;
    for(i=0; i<=n; i++)      r=r*b[i];
    return r;
}
void  main()
{
    int  x, a[]={2, 3, 4, 5, 6, 7, 8, 9};
    x=f(a, 3);
    printf("%d\n", x);
}
```

6.4　指针与字符串

6.4.1　使用指针表示字符串

在 C 程序中，字符串是一个字符序列，其开头用指针标识，末尾用空字符标识。

C 语言中常量包括字符串常量，字符串常量是用一对双引号括起来的字符序列。如："array"、"point"，在系统内部当做一个一维字符数组，在内存中连续存放。结尾存放字符串结束符，就是 ASCⅡ码为 0 的空字符'\0'。

数组名是数组的首地址，如“char sa[] = "array";”。

所以字符串常量实质上是一个指向该字符串首字符的指针常量，可以用字符指针表示字符串，如“char *sp = "point";”。

使用一维字符数组和字符指针都可以处理字符串，二者的区别如图 6.10 所示

例如：

```
char  sa[] = "This  is  a  string";
char  *sp = "This  is  a  string";
```

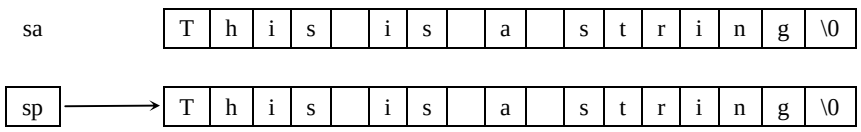


图 6.10　用字符数组和字符指针处理字符串示意图

字符数组是由若干连续的字符型数组元素构成的，字符指针存放的是字符串的首地址；字符数组的数组名是地址常量，其值不能改变，而字符指针是变量，其值是可以改变的。

如果要改变数组 sa 所代表的字符串，只能改变数组元素的内容。

如果要改变指针 sp 所代表的字符串，通常直接改变指针的值，让它指向新的字符串。

如图 6.10 所示，要把字符数组 sa 中存放的字符串改为"Hello"，对于用一维字符数组 sa

表示的字符串，就要使用字符串复制函数实现：

```
strcpy(sa, "Hello");
```

而对于用字符指针 sp 表示的字符串，只需要对指针 sp 赋值：“sp = "Hello";”此时 sp 就指向新字符串了。

但是，数组元素是变量，可以改变，如执行“sa[0]=sa[0]+32;”语句，sa[0]中的字符由'T'变成't'了。而字符指针所指向的是字符串常量，其中的字符不能改变，如*sp 指的是字符'T'，可以读出，但不能赋值，执行“printf("%c", *sp);”语句将输出字符'T'，而“*sp=*sp+32;”则是错误的。

【注意】①“sa = "Hello";”语句非法，因为数组名是常量，不能对它赋值。

② 使用字符指针时也要注意先赋值，后引用。指针在没有赋值前指向是不确定的。

例如：

```
char  *s ;
scanf("%s", s);          //错误，引用没有赋值的指针 s
```

应改为：

```
char  *s, str[20];
s = str;
scanf("%s", s);
```

同样的，可以在定义字符指针的同时赋空指针，如：

```
char  *s = NULL;
```

6.4.2 动态分配内存

在 C 语言中有以下两种方法获得内存空间：

① 程序编译时，编译系统为定义的各种类型的变量、数组分配相应大小的内存空间，程序编译完成后，这些变量、数组在内存中的大小就确定了，在程序的运行过程中不会改变。

② 在执行程序时，使用动态分配内存机制，利用一组库函数来实现。动态分配内存能够有效地利用存储空间，需要时就申请，使用完毕就释放，同一段内存在不同时刻可以有不同的用途。

动态分配内存函数与 void 类型指针密切相关。

void 类型是特殊的空类型，函数的返回值为 void 类型，表示函数无返回值。一般变量不能定义为 void 类型，但是指针变量可以定义为空类型，它能与所有的数据类型兼容。如一些库函数的返回值是 void 类型指针，使用时再用下述语句将其强制转换为其他类型的指针：

```
(数据类型名 *)void 型指针变量
```

所以 void 类型指针也称通用指针。

下面介绍一组动态内存分配库函数，使用这组库函数的程序需要包含头文件 stdlib.h。

1. **malloc()函数**

函数原型：void *malloc(unsigned size)

功能：在内存的动态存储区中分配一连续空间，其长度为 size。

① 若申请成功，则返回一个指向所分配内存空间的起始地址的基类型为 void 的指针；

② 若申请不成功，则返回 NULL(值为 0)；

③ 返回值类型为(void *)，这是通用指针的一个重要用途。

将 malloc()函数的返回值通过强制类型转换为特定指针类型，赋给一个指针，通过对这

个指针进行操作，就可以存取这个空间的数据。

例如：动态分配 50 个整数类型大小的空间的代码如下。

```
int *p;
p = (int *) malloc(50*sizeof(int))
if(p==NULL)
{   printf("Not able to allocate memory.\n");
    exit(1);
}
```

【说明】① 调用 malloc()时，用 sizeof 计算存储块大小以提高代码可移植性。

② 每次动态分配都要检查是否成功，考虑对例外情况进行处理。

③ 虽然存储块是动态分配的，但它的大小在分配后也是确定的，不要越界使用。

2. calloc()函数

函数原型：void *calloc(unsigned n, unsigned size)

在内存的动态存储区中分配 n 个连续空间，每一存储空间的长度为 size，并且分配后把存储块里内容全部初始化为 0。

① 若申请成功，则返回一个指向被分配内存空间的起始地址的基类型为 void 的指针；

② 若申请不成功，则返回 NULL。

3. free()函数

函数原型：void free(void *ptr)

释放由动态存储分配函数申请到的整块内存空间，ptr 为指向要释放空间的首地址。

当某个动态分配的存储块不再用时，要及时将它释放。

4. *realloc()函数

函数原型：void *realloc(void *ptr, unsigned size)

更改以前的存储分配，ptr 必须是以前通过动态存储分配得到的指针。参数 size 为现在需要的空间大小。

① 如果调整失败，返回 NULL，同时原来 ptr 指向存储块的内容不变；

② 如果调整成功，返回一片能存放大小为 size 的区块，并保证该块的内容与原块一致。如果 size 小于原块的大小，则内容为原块前 size 范围内的数据；如果新块更大，则原有数据存在新块的前一部分；

③ 如果分配成功，原存储块的内容就可能改变了，因此不允许再通过 ptr 去使用它。

【注意】使用以上函数时，需要引入头文件 stdlib.h，即程序头要有#include <stdlib.h>。

【例 6.14】动态分配一个能放置双精度数空间的程序示例。

参考代码：

```
#include<stdio.h>
#include<stdlib.h>
int main(void)
{
    double *p;
    p=(double*) malloc(sizeof(double));
    if(p==0)
```

```
    {
        printf("malloc error\n");
        exit(0);
    }
    *p=8.0;
    printf("*p=%f\n", *p);
    return  0;
}
```

运行实例：

```
*p=8.000000
```

在此例中，存放双精度数的空间不是在程序编译的时候分配的，而是通过调用 malloc()函数在运行程序时才分配的。

【练习 6.8】① 有以下程序：

```
#include<stdio.h>
#include<stdlib.h>
int  fun(int  n)
{
    int  *p;
    p=(int*)malloc(sizeof(int));
    *p=n;       return  *p;
}
void  main()
{
    int  a;
    a=fun(10);     printf("%d\n", a+fun(10));
}
```

运行程序的结果是_____。

A. 输出 0　　B. 输出 10　　C. 输出 20　　D. 出错

② 有以下程序段：

```
int  *p;
p= ____ malloc(sizeof(int));
```

若要使指针变量 p 指向一个存储整型变量的动态存储单元，则应填入____。

A. int　　B. int *　　C. (*int)　　D. (int*)

③ 若指针 p 已正确定义，int 型数据占 2 个字节，则下述使指针 p 指向两个连续整型动态存储单元的语句中，不正确的语句是____。

A. p=2*(int*)malloc(sizeof(int));　　B. p=(int*)malloc(2*sizeof(int));

C. p=(int*)malloc(2*2);　　D. p=(int*)calloc(2,sizeof(int));

6.4.3 常用的字符串处理函数

常用的字符串处理库函数包括：字符串的输入输出，字符串的复制，字符串的连接等。函数原型在 stdio.h 或 string.h 中给出。

1. 字符串的输入和输出

输入字符串：scanf()或 gets()。

输出字符串：printf()或 puts()。

函数原型在 stdio.h 中。

① scanf("%s", str)

输入参数：字符数组名，不加地址符。

遇回车或空格输入结束，并自动将输入的一串字符和 '\0'送入数组中

② gets(str)

遇回车输入结束，自动将输入的一串字符和 '\0'送入数组中

③ printf("%s", str) 或 printf("%s", "hello")

输出参数可以是字符数组名(如 str)或字符串常量(如"hello")，遇 '\0'结束输出。

④ puts(str) 或 puts("hello")

输出参数同上，输出字符串后自动换行。

【例 6.15】字符串输入输出示例。

```
#include<stdio.h>
int  main()
{
    char  str[80];
    scanf("%s", str);
    printf("%s", str);
    printf("%s", "World!");
    return  0;
}
```

运行实例 1：

```
Hello
HelloWorld!
```

运行实例 2：

```
How  are  you!
HowWorld!
```

使用 gets()和 puts()函数，修改程序如下：

```
#include<stdio.h>
int  main()
{
    char  str[80];
    gets(str);
    puts(str);
    puts("World!");
    return  0;
}
```

运行实例 3：

```
Hello
Hello
World!
```

运行实例 4：

```
How  are  you!
How  are  you!
World!
```

2. 专门的字符串处理函数

在 string.h 中定义了若干专门用来处理字符串的函数。

① 字符串拷贝函数 strcpy()

格式：strcpy(目的字符串，源字符串);

功能：把源字符串中的字符串拷贝到目的字符串中。串结束标志 '\0' 也一同拷贝。目的字符串是字符数组名，源字符串可以是字符数组名也可以是一个字符串常量。这时相当于把一个字符串常量赋值给字符数组。

字符串不能整体赋值，只能使用字符串复制函数。

例如要将字符串 str2 复制到 str1 中：

```
str1 = str2;                //错误，str1 是数组名，表示常量，不能被赋值
strcpy(str1, str2);         //正确
```

【例 6.16】字符串复制示例。

```
# include<stdio.h>
# include<string.h>
int  main(void)
{
    char  str1[20], str2[20];
    gets(str2);
    strcpy(str1, str2);
    puts(str1);
    return  0;
}
```

运行实例：

```
How  are  you!
How  are  you!
```

② 字符串拼接函数 strcat()

格式：strcat(目的字符串，源字符串);

功能：把源字符串中的有效字符拼接到目的字符串中的有效字符后面，新字符串放在目的字符串的字符数组中，串结束标志 '\0' 在新字符串的末尾。目的字符串是字符数组名，源字符串可以是字符数组名也可以是字符串常量。函数返回值是目的字符串的首地址。例如：

```
strcat(str1, str2);
```

连接两个字符串 str1 和 str2，并将结果放入 str1 中，注意对 str1 必须分配足够大的空间。

【例 6.17】字符串拼接示例。

```
#include<stdio.h>
#include<string.h>
int  main(void)
{
    char  str1[80], str2[20];
    gets(str1);
    gets(str2);
    strcat(str1, str2);
```

```
    puts(str1);
    return 0;
}
```

运行实例：

```
How are
  you!
How are   you!
```

③ 字符串比较函数 strcmp()

格式：strcmp(字符串 1, 字符串 2)；

功能：按照 ASCII 码顺序比较两个字符串，并用函数返回值返回比较结果。

- ❖ 若字符串 1 和字符串 2 相等，返回 0。
- ❖ 若字符串 1 大于字符串 2，返回一个正整数+1。
- ❖ 若字符串 1 小于字符串 2，返回一个负整数-1。

字符串不能整体比较大小，例如：

```
if(str1 > str2)                    //错误，这样比较的是两个字符串首地址值的大小
if(strcmp(str1, str2) > 0)         //正确
```

【例 6.18】字符串大小比较示例。

```
# include<stdio.h>
# include<string.h>
int main(void)
{
    int res;
    char s1[20], s2[20];
    gets(s1);
    gets(s2);
    res = strcmp(s1, s2);
    printf("%d", res);
    return 0;
}
```

运行实例 1：

```
run
red
 1
```

运行实例 2：

```
run
running
-1
```

④ 求字符串长度函数 strlen()

格式：strlen(字符串)

功能：计算字符串的有效长度，不包括字符串结束符 '\0'，并将其作为函数返回值，返回值类型为 unsigned 类型，strlen()也接受一个字符指针参数，返回值为参数所指的位置和字符串结束符之间的字符数。例如：

```
static char str[20]= "How are you?"
strlen("hello")的值是：5
strlen(str)的值是：12
```

字符串处理函数的小结如表 6.2 所示。

表 6.2　字符串处理函数小结

函　数	功　能	头文件
puts(str)	输出字符串，结尾加回车	stdio.h
gets(str)	输入字符串(以回车结束)	
strcpy(s1, s2)	将字符串 s2 复制到字符串 s1，返回 s1	string.h
strcat(s1, s2)	将字符串 s2 拼接到字符串 s1 后面，结束符移到新串后，返回 s1	
strcmp(s1,s2)	按 ASCII 码值比较两个字符串的对应的字符 若两个字符串相等，返回值为 0 若字符串 s1 大，返回值>0 (+1) 若字符串 s1 小，返回值<0 (−1)	
strlen(str)	计算字符串的有效长度，不包括'\0'，返回值为 unsigned 类型	

【练习 6.9】① 若有定义语句“char s[10]="1234567\0\0";”，则 strlen(s)的值是_____。

A. 7　　B. 8　　C. 9　　D. 10

② 有以下程序：

```
#include<stdio.h>
#include<string.h>
void  main( )
{
    char  a1[]="abcd", a2[]="ABCD", *p1=a1, *p2=a2, str[50]="xyz";
    strcpy(str+2, strcat(p1+2, p2+1));
    printf("%s", str);
}
```

运行程序后的输出结果是_____。

A. xyabcAB　　B. abcABz　　C. ABabcz　　D. xycdBCD

③ 以下语句或语句组中，能正确进行字符串赋值的语句是_____。

A. char *sp; *sp="right!";　　B. char s[10]; s="right!";

C. char s[10]; *s="right!";　　D. char *sp="right!";

④ 有以下程序：

```
void  ss(char  *s, char  t)
{
    while(*s)
    {
        if(*s==t)
            *s=t-'a'+'A';
        s++;
    }
}
void  main()
{
```

```
    char  str1[100]="abcddfefdbd",        c='d';
    ss(str1, c);
    printf("%s\n", str1);
}
```

运行程序后的输出结果是_____。

A. ABCDDFEFDBD　B. abcDDfefDbD　C. abcAAfefAbA　D. Abcddfefdbd

⑤ 有以下程序：

```
#include<stdio.h>
#include<string.h>
void  main()
{
    char  *s1="AbCdEf",   *s2="aB";
    s1++;   s2++;
    printf("%d\n", strcmp(s1, s2));
}
```

运行程序后的输出结果是_____。

A. 正数　　B. 负数　　C. 零　　D. 不确定的值

⑥ 有以下程序：

```
#include<stdio.h>
f(char  *s)
{   char  *p=s;
    while(*p!='\0')     p++;
    return(p-s);
}
void  main()
{
    printf("%d\n", f("ABCDEF"));
}
```

运行程序后的输出结果是_____。

A. 3　　B. 6　　C. 8　　D. 0

⑦ 已定义如下函数：

```
fun(char  *p2,char  *p1)
{while((*p2=*p1)!='\0')   {p1++;    p2++;}}
```

函数的功能是____。

A. 将p1所指的字符串复制到p2所指的内存空间

B. 将p1所指的字符串的地址赋给指针p2

C. 对p1和p2两个指针所指的字符串进行比较

D. 检查p1和p2两个指针所指的字符串中是否有'\0'

⑧ 有以下程序：

```
#include<stdio.h>
#include<string.h>
```

```
main()
{   char ss[10]="12345";
    strcat(ss, "6789");
    gets(ss);    printf("%s\n", ss);
}
```

当执行上述程序且输入ABC时，输出的结果是_____。

A. ABC　　B. ABC9　　C. 123456ABC　　D. ABC456789

【练习 6.10】运行以下程序后的输出结果是____。

```
#include<string.h>
void fun(char *s, int p, int k)
{   int i;
    for(i=p; i<k-1; i++)   s[i]=s[i+2];
}
void main()
{   char s[]="abcdefg";
    fun(s, 3, strlen(s));
    puts(s);
}
```

【练习 6.11】运行以下程序后的输出结果是____。

```
void main()
{   char a[]="language";
    char *ptr=a;
    while(*ptr)
    {   printf("%c", *ptr-32);
        ptr++;
    }
}
```

6.5 指针进阶

6.5.1 二级指针

前面已经介绍过，指针是一个变量，其值是另一个变量的地址。可以使用地址运算符&让指针指向某个变量，例如：

```
int a, *ptr;
ptr=&a;
```

上面的程序段定义了整型变量 a 和整型指针变量 ptr，然后让 ptr 指向变量 a。

在此后执行“a=9;”或“*ptr=9;”，都可以将 9 赋给变量 a。

由于指针本身是一个数值变量，也存储在计算机内存的单元中，因此可以创建指向指针的指针——二级指针变量，其值是一个指针的地址。

定义指针的指针时，使用两个**，同样可以使用**来存取被指向的变量。

例如下面代码声明了指向指针的指针 ptr2，其指向如图 6.11 所示。

```
int  a=12;
int  *ptr=&a;
int  **ptr2=&ptr;
```

指向指针的指针常用在指针数组中。

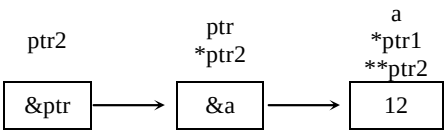


图 6.11　指向指针的指针

6.5.2　指针与二维数组

C 语言中定义的二维数组，可以看成是一个一维数组，这个一维数组的每一个数组元素又是一个一维数组。C 语言定义二维数组时使用两对方括号，例如：

```
int  a[3][4];
```

定义了一个名为 a 的数组，包含 3 个元素，a[0]、a[1]、a[2]，而 a[0]、a[1]、a[2]又分别各是一个包含 4 个元素的一维数组的数组名。数组名 a 代表了整个二维数组的首地址，是第一个数组元素 a[0]的地址，a+1 是第二个数组元素 a[1]的地址，a+2 是第三个数组元素 a[2]的地址。如图 6.12 所示。

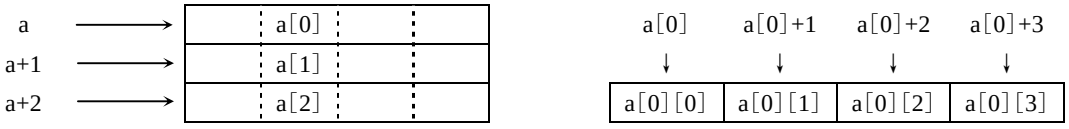


图 6.12　指向一维数组的指针

a[0]也是一维数组的数组名，是第一个元素 a[0][0]的地址，该数组里包含 a[0][0]、a[0][1]、a[0][2]、a[0][3]四个整型变量。同样的，a[1]、a[2]也是数组名，分别是其第一个数组元素的地址。

对于一个二维数组来说，有 2 对方括号的是数组元素，如 a[0][0]、a[1][2]、a[2][0]等，而少于 2 对方括号的 a、a[0]、a[1]、a[2]都是指针。

可以定义一个指向二维数组一行的指针变量，一般格式如下：

基类型 (*指针变量名)[长度]

含义：定义一个指针变量，使其指向二维数组的一整行，定义中的圆括号不能省略。

例如“int (*p)[4];”定义一个指针变量 p，该指针变量可以指向一个由 4 个 int 元素组成的一维数组。

因此，可以使 p 指向上例中的二维数组中的某一行，p=a[0]，指向第一行；p+1 指向第二行，即 a[1]；p+2 指向第三行，即 a[2]。

6.5.3　指针数组

数组是一组具有相同类型数据构成的数据结构，数组元素在内存中连续存放。指针也是一种数据类型，因此也可以构造指针数组，指针数组的功能很强大，常用于处理多个字符串。

字符串是一个存储在内存中的字符序列，其开始位置由指向其第一个字符的 char 类型的指针标识，结尾有字符串结束符来标记。使用 char 指针数组可以处理多个字符串，数组中的每个元素(char 指针)都指向一个不同的字符串，通过遍历整个数组可以依次存取多个字符串。

定义 char 类型指针数组：

```
char  *color[10];  //定义一个包含 10 个元素的 char 类型指针数组
```

数组 color 的每个元素都是一个 char 指针。同样的可以在定义的同时初始化，例如：

```
char  *color[10]={"red", "blue", "yellow", "green", "purple"};
```

该语句完成功能如下：

① 给数组 color 分配连续的存储空间，存放 10 个数组元素，每个元素都是 char 指针，如图 6.13 所示；

② 在内存的空闲处分配空间，存放 5 个初始字符串，每个字符串以字符串结束符结束；

③ 初始化数组元素 color[0]、color[1]、color[2]、color[3]、color[4]，使 5 个 char 指针分别指向第 1 个字符串"red"、第 2 个字符串"blue"、第 3 个字符串"yellow"、第 4 个字符串"green"、第 5 个字符串"purple"的第一个字符的地址。

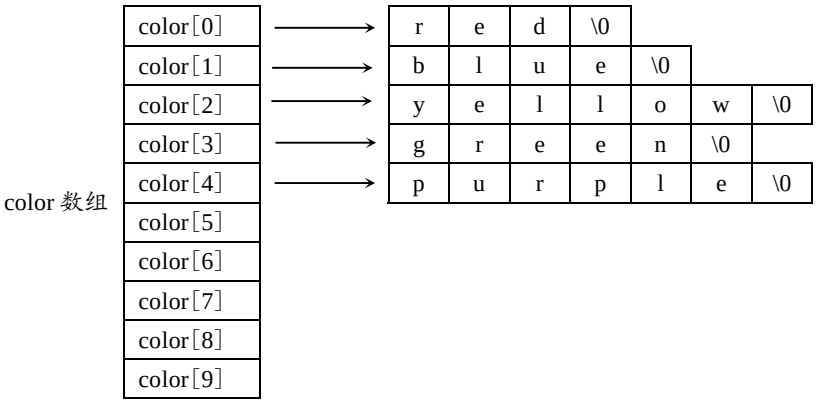


图 6.13　指针数组

【例 6.19】指针数组使用示例。

```
#include<stdio.h>
int  main(void)
{
    int  i;
    char  *color[5] = {"red", "blue", "yellow", "green", "purple"}, *tmp;
    for(i = 0; i < 5; i++)
       printf("%x, %s\n", color[i], color[i]);        // 输出 5 个字符串的地址和内容
    tmp = color[0];                                   // 交换 color[0]与 color[4]
    color[0] = color[4];
    color[4] = tmp;
    printf("color[0]: %s, color[4]:%s\n", color[0], color[4]);
    return  0;
}
```

运行实例(实际显示的地址信息会随系统不同而不同)：

```
420068,  red
420060,  blue
420058, yellow
420050, green
420048, purple
color[0]: purple, color[4]:red
```

【注意】指针交换的结果是指针 color[0]指向了 purple，指针 color[4]指向了 red，如图 6.14 所示。

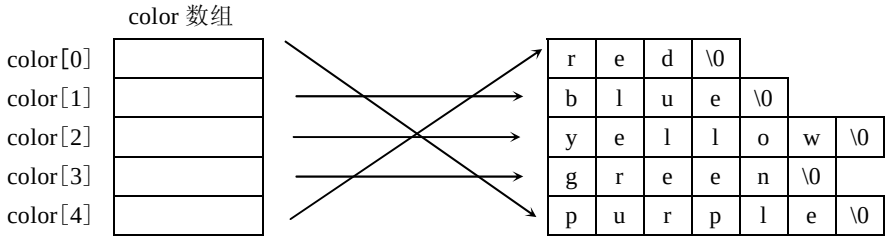


图 6.14　使用指针数组

【例 6.20】从键盘上输入 6 个字符串，每个字符串长度在 20 个字符以内。排序后输出。用函数调用实现。

```
#include<stdio.h>
# include<stdlib.h>
#include<string.h>
void sort(char *p[], int n);          //函数声明
int main(void)
{
    int i;
    char *line[6], str[20];           //定义指针数组用来处理多个字符串
    printf("Input 6 strings:\n");
    for(i = 0; i < 6; i++)            //输入字符串
    {
        gets(str);
        if((line[i]=(char *)malloc(strlen(str)+1))==NULL)   //动态申请存储空间
            return -1;
        strcpy(line[i], str);                 //字符串复制
    }
    sort(line, 6);                            // 调用排序函数对字符串排序
    printf("After sort:\n");
    for(i = 0; i < 6; i++)                    // 输出排序后的字符串
        puts(line[i]);
  return 0;
}
void sort(char *p[], int n)                   // 用选择法对多个字符串排序
{
  int i, j, index;
  char *temp;
  for(i=0; i<n-1; i++)
      {   index=i;
          for(j=i+1; j<n; j++)
          if(strcmp(p[index], p[j])>0)   //比较 2 个字符串大小
```

```
                index=j;
                temp=p[index];
                p[index]=p[i];
                p[i]=temp;
            }
        }
```

运行实例：

```
Input 6 strings:
one
two
three
four
five
six
After sort:
five
four
one
six
three
two
```

【练习 6.12】① 设有定义：

```
int n=0, *p=&n, **q=&p;
```

以下赋值语句中，正确的语句是_____。

A. p=1;　　B. *q=2;　　C. q=p;　　D. *p=5;

② 设有语句“int (*ptr)[M];”，其中的标识符ptr是_____。

A. M个指向整型变量的指针

B. 指向M个整型变量的函数指针

C. 一个指向M个整型元素的一维数组的指针

D. 具有M个指针元素的一维指针数组，每个元素都只能指向整型变量

③ 设有语句“static int a[2][3]={2, 4, 6, 8, 10, 12};”，下述各项中，正确表示数组元素地址的是_____。

A. *(a+1)　　B. *(a[1]+2)　　C. a[1]+3　　D. a[0][0]

④ 有以下程序：

```
#include<stdio.h>
void main()
{
    char *alpha[6]={"ABCD", "EFGH", "IJKL", "MNOP", "QRST", "UVWX"};
    char **p;
    int i;
    p=alpha;
```

```
        for(i=0; i<4; i++)
            printf("%s",p[i]);
        printf("\n");
    }
```

运行程序后的输出结果为_____。

A. ABCDEFGHIJKL　　B. ABCD

C. ABCDEFGHIJKLMNOP　　D. AEIM

⑤ 有以下程序：

```
void main()
{   char *s[]={"one", "two", "three"}, *p;
    p=s[1];
    printf("%c, %s", *(p+1), s[0]);
}
```

运行程序后的输出结果为_____。

A. n, two　　B. t, one　　C. w, one　　D. o, two

【练习 6.13】运行以下程序后的输出结果是_____。

```
void main()
{   int a[5]={2, 4, 6, 8, 10}, *p, **k;
    p=a;
    k=&p;
    printf("%d, ", *p++);
    printf("%d\n", **k);
}
```

【练习 6.14】运行以下程序后的输出结果是_____。

```
#include<string.h>
void main()
{   char ch[]="abc", x[3][4];
    int i;
    for(i=0; i<3; i++)    strcpy(x[i], ch);
    for(i=0; i<3; i++)    printf("%s", &x[i][i]);
}
```

6.5.4　命令行参数

C 语言源程序经编译和连接处理，生成可执行文件后，才能运行。

在 DOS 环境的命令窗口中，输入可执行文件名，就可以按命令方式运行该程序。

输入命令时，在可执行文件(命令)名的后面可以跟一些参数，这些参数被称为命令行参数。格式如下：

命令名 参数 1　参数 2 … 参数 n

命令名和各个参数之间用空格分隔，也可以没有参数。

使用命令行的程序不能在编译器中运行，需要将源程序经编译、连接为相应的命令文件(一般以.exe 为后缀)，然后回到命令行状态，再在该状态下直接输入命令文件名。

【例 6.21】带参数的 main()函数示例。

```
#include<stdio.h>                    //源程序文件名为 test.c
int main(int argc, char *argv[])
{
    printf("Hello ");
    printf("%s", argv[1]);
    return 0;
}
```

在 DOS 命令行状态下，以命令的方式输入

```
test world!
```

其中 test 是命令，world!是参数，屏幕显示：

```
Hello world!
```

主函数 main(int argc, char *argv[])有两个参数。

第 1 个参数 argc 接收命令行参数(包括命令名)的个数，在例 6.21 中 argc=2。

第 2 个参数*argv[]接收以字符串常量形式存放的命令行参数(命令名本身也作为一个参数)，字符指针 argv[0]指向命令名，例 6.21 中为 test；argv[1]指向第一个命令行参数，例 6.21 中为 world。

【例 6.22】编写 C 程序 echo，它的功能是将所有命令行参数(不包括命令名)在同一行上输出。

```
#include<stdio.h>                          //源程序文件名为  echo.c
int main(int argc, char *argv[])
{
    int k;
    for(k = 1; k < argc; k++)              // 不包括命令名，所以 k 初值为 1
    printf("%s ", argv[k]);                // 输出命令行参数
    printf("\n");
    return 0;
}
```

命令行状态下，输入：

```
echo How are you?
```

此时，main()函数的 2 个参数分别为：argc=4，*argv[]：{"echo", "How","are","you"}。屏幕显示为：

```
How are you?
```

【练习 6.15】① 有以下程序：

```
#include<string.h>
main(int argc, char *argv[])
{
    int i, len=0;
    for(i=1; i<argc; i+=2)     len+=strlen(argv[i]);
    printf("%d", len);
}
```

经编译链接后生成的可执行文件是ex.exe，若运行时输入以下带参数的命令行

ex abcd efg h3 k44

运行程序后的输出结果是____。

A. 14　　B. 12　　C. 8　　D. 6

② 假定下列程序的可执行文件名为prg.exe：

```
main(int argc, char *argv[])
{   int i;
    if(argc<=0)return;
    for(i=1; i<argc; i++)      printf ("%c", *argv[i]);
}
```

则在该程序所在的子目录下输入命令行：

prg hello good

按回车后，输出结果是____。

A. hello good　　B. hg　　C. hel　　D. hellogood

6.5.5 返回指针的函数

返回指针的函数是指函数返回值的类型是指针类型，即返回一个地址。函数的定义、调用方法与其他函数一样。

【例 6.23】输入一个字符串和一个字符，如果该字符在字符串中，就从该字符首次出现的位置开始输出字符串中的字符。例如，输入字符串 program 和字符 r 后，输出 rogram。

要求定义函数 match(s, ch)，在字符串 s 中查找字符 ch，如果找到，返回第一次找到的该字符在字符串中的位置(地址)；否则，返回空指针 NULL。

参考代码：

```
#include<stdio.h>
char *match(char *s, char ch)          // 函数返回值类型是指向 char 类型的指针
{
   while(*s != '\0')
   if(*s == ch)    return(s);          //若找到字符 ch，返回相应的地址
   else            s++;
   return(NULL);                       //没有找到 ch，返回空指针
}
int main()
{
    char ch, str[80], *p = NULL;
    printf("Please input the string:\n");
    scanf("%s", str);
    getchar();                         // 跳过输入字符串和输入字符之间的分隔
    ch = getchar();
    if((p = match(str, ch)) != NULL)   //函数返回的指针赋给 char 类型指针 p
        printf("%s\n", p);
    else
```

```
            printf("Not  Found\n");
        return  0;
    }
```

运行实例 1：

```
Please  input  the  string:
Program  r
rogram
```

运行实例 2：

```
Please  input  the  string:
Program   y
Not  Found
```

6.5.6 指向函数的指针

每个函数的定义经过编译后都占用一段内存单元，有一个入口地址(起始地址)。函数名就代表函数的入口地址。

函数指针：一个指针变量，接收函数的入口地址，让它指向函数，如图 6.15 所示。

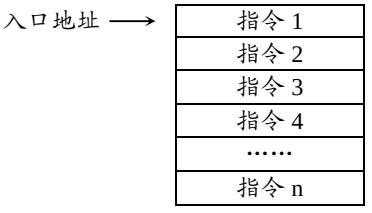


图 6.15　指向函数的指针

定义了函数指针并让它指向某个函数后，就可以通过函数指针调用函数。函数指针还可以作为函数的参数。

1. 函数指针的定义

```
类型名 (*变量名)();
```

类型名即函数指针所指向的函数的返回值的类型。例如：

```
int (*funptr)();
```

定义一个函数指针 funptr，funptr 指向一个返回值类型为 int 的函数。

2. 函数指针的赋值

```
函数指针名=函数名;
```

例如：

```
int  fun(x, y)          //函数定义，函数的返回值是 int 类型
{
    return  x > y ? x : y;
}
funptr = fun;
```

上述程序段最后的赋值语句将 fun()函数的入口地址赋给变量 funptr，即 funptr 指向函数 fun()。

3. 函数指针的使用

函数指针的使用主要是通过函数指针调用函数，格式如下：

```
(*函数指针名)(参数表)
```

普通调用函数的语句是用函数名直接调用，例如：

```
z = fun(3, 5);
```

也可以通过函数指针调用，函数指针必须已经定义并赋值，如：

z = (*funptr)(3, 5);

这两种调用方式结果完全一样。

本章介绍了指针的多种用法，C 语言中指针的使用很灵活，也很有效，但是也给初学者带来一定困难，表 6.3 是有关指针的数据类型的小结，为便于比较，相关的其他数据类型也列在一起。

表 6.3　有关指针的数据类型的小结

定　　义	含　　义
int *p;	定义指向整型变量的指针变量 p
int a[10];	定义整型数组 a，该数组有 10 个元素
int *p[10];	定义指针数组 p，它由 10 个指向整型变量的指针元素组成
int (*p)[10];	定义指针变量 p，指向含有 10 个整型元素的一维数组
int f();	定义函数，f()的返回值类型是整型数值，无参数
int *f();	定义函数，f()的返回值为指向整型数据的指针，无参数
int (*p)();	定义函数指针变量 p，p 指向返回值为整型数值的函数
int **p;	定义二级指针变量 p，p 指向基类型为整型数据的指针

【练习 6.16】① 若有说明语句“double *p, a;”则下述各程序段中，能通过语句正确给输入项读入数据的程序段是_____。

A. *p=&a;　scanf("%1f", p);　　B. *p=&a;　scanf("%f", p);

C. *p=&a;　scanf("%1f", *p);　　D. p=&a;　scanf("%1f", p);

② 有以下程序：

```
void swap(char *x, char *y)
{   char t;
    t=*x;    *x=*y;      *y=t;
}
void main()
{   char s1[]="abc", s2[]="123";
    swap(s1, s2);   printf("%s,%s\n", s1, s2);
}
```

运行程序后的输出结果是_____。

A. 123, abc　　B. abc, 123　　C. 1bc, a23　　D. 321, cba

③ 有以下程序：

```
void main()
{
   char s[]={"aeiou"}, *ps;
   ps=s;      printf("%c\n", *ps+4);
}
```

运行程序后的输出结果是_____。

A. a　　B. e　　C. u　　D. 元素s[4]的地址

④ 有以下程序：

```
int *f(int *x, int *y)
```

```
{   if(*x<*y)
        return  x;
    else
        return  y;
}
void  main()
{
    int  a=7, b=8, *p, *q, *r ;
    p=&a;     q=&b;
    r= f(p, q);
    printf("%d, %d, %d", *p, *q, *r);
}
```

运行程序后的输出结果是_____。

A. 7, 8, 8　　B. 7, 8, 7　　C. 8, 7, 7　　D. 8, 7, 8

⑤ 定义语句“int (*ptr)();”的含义是_____。

A. ptr 是指向一维数组的指针变量

B. ptr 是指向 int 型数据的指针变量

C. ptr 是指向函数的指针，该函数返回一个 int 型数据

D. ptr 是一个函数名，该函数的返回值是指向 int 型数据的指针

⑥ 已有定义“int (*p)();”指针 p_____。

A. 代表函数的返回值　　B. 指向函数的入口地址

C. 表示函数的类型　　D. 表示函数的返回值的类型

⑦ 已有函数 max(a, b)，为了让函数指针变量 p 指向函数 max，正确的赋值方法是_____。

A. p=max;　　B. *p=max;　　C. p=max(a,b);　　D. *p=max(a,b);

⑧ 已有函数 max(a, b)，函数指针变量 p 已经指向函数 max，正确调用该函数的方法是_____。

A. (*p)=max(a,b);　　B. *pmax(a,b);　　C. (*p)(a, b);　　D. *p(a, b);

习　题　6

编写程序，实现下述各题的要求。

1. 从键盘输入 5 个整数，按照从小到大的顺序排序，然后输出。排序用子函数实现。

2. 从键盘输入 5 个字符串，按照从小到大的顺序排序，然后输出。排序用子函数实现。

3. 输入一个字符串，再输入一个字符 ch，将字符串中包含的所有 ch 字符删除后输出该字符串。定义和调用 delchar(s, c)函数来实现，delchar(s, c)函数的功能是删除字符串 s 中出现的所有字符 c。

4. 判断输入的一串字符是否是回文。回文就是顺读和倒读都一样的字符串。

5. 输入一行文字，统计其中的大写字母、小写字母、空格、数字以及其他字符各有多少。

6. 将字符串 programming 赋给一个字符数组，然后从第一个字母开始间隔地输出该字符串，用指针编程实现。

7. 编写函数，将字符串中的第 m 个字符开始的全部字符复制成另一个字符串。要求在主函数中输入字符串及 m 的值并输出复制结果，在被调函数中完成复制。

8. 编程实现将字符串 str1 的所有字符传送到字符串 str2 中，要求每传送 3 个字符后再存放一个空格，例如字符串 str1 为“abcdefg”，则字符串 str2 为“abc def g”。

第 7 章　构造数据类型与编译预处理

C 语言具有丰富的数据类型，我们已经学习了基本数据类型，包括整型、浮点型、字符型、指针类型、空类型，还学习了数组，数组属于构造数据类型，所谓构造数据类型就是程序员根据已经定义的数据类型用构造的方法定义的新的数据类型，C 语言中构造数据类型包括数组、结构体、共用体以及枚举类型。本章主要介绍结构体、共用体、枚举类型的用法。

C 语言程序中可以包含以“#”开始的编译指令，这些指令称为预处理命令，预处理命令由编译系统的预处理程序负责处理。本章主要介绍三类常用的预处理命令：宏定义、文件包含和条件编译。

知 识 结 构

1. **结构体**
   ① 结构体类型的定义
   ② 结构体变量的定义、初始化与引用
2. **结构体数组**
3. **链表**
4. **共用体**
   共用体类型的定义、变量定义与引用
5. **枚举类型**
6. **编译预处理命令**
   ① 宏定义
   ② 文件包含
   ③ 条件编译

7.1　结　构　体

现实生活中经常会遇到这种问题，几个数据之间有密切的联系，它们用来描述一个事物的几个方面，但是它们并不属于同一数据类型。例如，前面提到的学生记录中的各个数据项描述了一个学生的几个不同侧面。如果用几个独立的变量来表示，很难看出这些数据项之间的联系，处理起来不方便。C 语言提供了一种可以把不同类型的数据项(当然也可以是相同类型数据项)组成一个整体的数据类型，这就是结构体类型。学生记录用结构体类型表示如图 7.1 所示，它构成了一种新的数据类型。

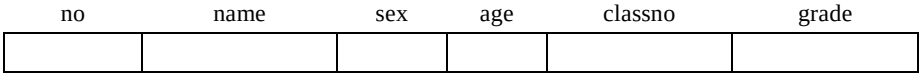


图 7.1　用结构体类型表示学生信息

7.1.1　结构体类型的定义

不同的数据项可以构成不同的结构体类型，因此结构体类型在使用之前必须先定义。前面讨论的学生信息，用结构体类型描述如下：

```
struct  Student
{
    char  no[10];          //学号
    char  name[20];        //姓名
    char  sex;             //性别
    int  age;              //年龄
    int  classno;          //班级
    float  grade;          //成绩
};
```

这里定义了一个 struct Student 结构体类型，其中 struct 是关键字，用它来标志结构体类型，Student 是结构体名。

结构体类型定义的格式如下：

```
struct  结构体名
{
    数据类型1  结构体成员变量名1;
    数据类型2  结构体成员变量名2;
    ……
    数据类型n  结构体成员变量名n;
};
```

【说明】① struct 是关键字，表示定义一个结构体类型，不能省略。

② 结构体名必须是合法的自定义标识符，不能是关键字。

③ 成员变量的数量和类型不限，类型可以是基本数据类型，也可以是构造数据类型，成员变量间的顺序也不限。

④ 整个结构体的定义必须以分号结尾。

例如，下面定义了一个描述日期的 struct Date 结构体类型。

```
struct  Date
{
    int  year;          //年
    int  month;         //月
    int  day;           //日
};
```

在结构体中数据类型相同的成员，既可逐个、逐行分别定义，也可合并成一行定义，就像一次定义多个变量一样。前面的 struct Student 和 struct Date 结构体类型定义也可以写成下面的形式。

```
struct  Student
{
    char  no[10], name[20], sex;        //学号，姓名，性别
    int  age, classno;                  //年龄，班级
    float  grade;                       //成绩
};
struct  Date
{
    int  year, month, day;              //年，月，日
};
```

7.1.2 结构体变量的定义

与系统定义的基本类型(如 int、char、float 等)一样，也可以用结构体类型来定义变量，这种变量称之为结构体变量。定义结构体变量的方法，可概括为如下三种方法：

1. 先定义结构体类型，再定义结构体变量

```
struct  结构体名
{
    数据类型1  结构体成员变量名1;
    数据类型2  结构体成员变量名2;
    ……
    数据类型n  结构体成员变量名n;
};
struct  结构体名  变量名表;
```

可以利用 7.1.1 节定义的学生信息结构体类型 struct Student 定义一个相应的结构体变量 student1，定义语句如下：

```
struct  Student  student1;
```

定义这个结构体变量时，系统将为这个变量分配 sizeof(struct Student)大小的存储空间，并且按照结构体类型定义中成员定义的顺序为各个成员变量分配内存空间，如图 7.2 所示。

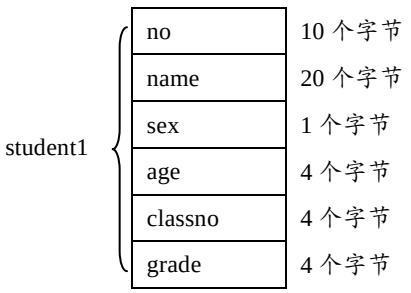


图 7.2　结构体变量内存空间分配图

结构体是一种构造类型，它由多个成员变量组合而成。因此这种类型的变量所占内存的大小是它所包含的成员变量所占内存大小的和。对于 struct Student 来说，它所定义的变量所占内存的字节数为：

```
sizeof(struct  Student)
=sizeof(no)+sizeof(name)+sizeof(sex)+sizeof(age)+sizeof(classno)+sizeof(grade)
```

【注意】结构体类型只是用户自定义的一种数据类型，同前面所介绍的基本数据类型一样，它本身不占用内存单元，只有用它来定义某个变量时，才会为该变量分配结构体类型所需要大小的内存单元。

在编程的时候，根据实际情况需要，也可以同时定义多个结构体类型变量，变量间用逗

号分开。

```
struct  Student  student1, student2;
```

此外，也可以定义指向结构体类型的指针变量，如：

```
struct  Student  *p;
```

定义指针变量 p，p 是指向 struct Student 结构体类型的指针变量。

2. 在定义结构体类型的同时，定义结构体变量

```
struct  结构体名
{
    数据类型1  结构体成员变量名1;
    数据类型2  结构体成员变量名2;
    ……
    数据类型n  结构体成员变量名n;
}变量名列表;
```

例如：

```
struct  Student
{
    char  no[10];              //学号
    char  name[20];            //姓名
    char  sex;                 //性别
    int  age;                  //年龄
    int  classno;              //班级
    float  grade;              //成绩
 }student1, student2;
```

以上实例既定义了 struct Student 结构体类型，又同时定义了 2 个该类型的变量 student1 和 student2。

3. 直接定义结构体变量

```
struct
{
    数据类型1  结构体成员变量名1;
    数据类型2  结构体成员变量名2;
    ……
    数据类型n  结构体成员变量名n;
}变量名列表;
```

例如下述程序段：

```
struct
{
    char  no[10];              //学号
    char  name[20];            //姓名
    char  sex;                 //性别
    int  age;                  //年龄
    int  classno;              //班级
```

```
    float grade;                    //成绩
}student1, student2;
```

第 3 种方法与第 2 种方法的区别：第 3 种方法中省去了结构体名，而直接给出了结构体变量，采用这种方法无法使用方法 1 进行定义结构体变量，因为此时缺少结构体名。

【说明】① 结构体类型与结构体变量是两个不同的概念，其区别和 int 类型与 int 型变量的区别一样。结构体类型不分配内存，而结构体变量分配内存；结构体类型不能赋值、存取和运算，而结构体变量则可以。

② 结构体类型定义可以嵌套。即结构体中的成员可以是另一结构体类型的变量，也可以是自身类型的指针，但不能是自身类型的变量。

例如，在前面的学生信息结构体中如果将年龄改为生日，则可以用下面的结构体类型来定义学生信息，此时某些结构体成员也是一个结构体类型的变量。如下所述：

```
struct Date
{
    int month;
    int day;
    int year;
};
struct
{
    char no[10];
    char name[20];
    char sex;
    struct Date birthday;
    int classno;
    float grade;
}student1, student2;
```

上述定义中，定义了一个 struct Date 结构体，它由 month(月)、day(日)、year(年)3 个成员变量组成。在定义变量 student1，student2 时，其中的成员 birthday 被说明为 struct Date 结构体类型。

③ 成员变量名可与程序中其他变量同名，它们之间互不干扰。

```
char sex;                   //程序中的变量
struct
{
    char no[10], name[20];
    char sex;               //结构体类型中的成员变量
    int age, classno;
    float grade;
 }student1, student2;
```

【练习 7.1】以下结构体类型变量 student1 占用多少个字节内存空间？

```
struct Student
{
```

```
    char  no[10], name[20], sex;        //学号，姓名，性别
    int  age, classno;                  //年龄，班级
    float  grade;                       //成绩
}student1;
```

7.1.3　结构体变量的引用

引用结构体变量的时候应遵循以下规则：

① 不能对一个结构体变量作为一个整体进行输入输出。只能对结构体变量中的各个成员变量分别进行输入输出。引用方式为：

结构体变量名.成员名

例如：

```
scanf("%c", &student1.sex);
```

对于结构体指针变量来说，一般通过“->”运算符来访问其成员，也可以用“.”运算符来访问，其访问的一般格式为：

结构体指针->成员名　　或　　(*结构体指针).成员名

“.”和“->”是成员引用运算符，它们在所有运算符中优先级最高，所以可以把 student1.sex 或 student1->sex 看做一个整体。

② 可以使用赋值运算符将一个结构体变量赋值给另外一个相同类型的结构体变量。例如：

```
student1=student2;
```

表示将 student2 的所有成员变量的值分别赋值给 student1 对应的每个成员变量。

③ 如果成员变量本身又属于一个结构体类型，则要用若干个成员运算符，一级一级地找到最低的一级的成员，只能对最低级的成员进行赋值或存取运算。例如：

```
student1.birthday.month=10;
scanf("%d", &student1.birthday.day);
```

④ 对成员变量可以像普通变量一样进行符合其类型的各种运算。例如：

```
student1.sex=student2.sex;
strcpy(student1.name, "zhangsan");
sum= student1.grade+student2.grade;
```

7.1.4　结构体变量的初始化

对结构体变量初始化的时候要对每个成员进行初始化，例如：

```
struct  Student
{
    char  no[10];              //学号
    char  name[20];            //姓名
    char  sex;                 //性别
    int  age;                  //年龄
    int  classno;              //班级
    float  grade;              //成绩
}student1={"102", "zhangsan", 'M', 21, 12, 98.4};
```

初始化时，将所提供的数据按照各成员变量的顺序排列，如果成员变量仍是结构体类型，

则按最低层类型提供数据。

另外，还可以通过分别赋值的方式为结构体变量的各个成员变量赋值，例如：

```
student1.age=24;
```

下面举例说明结构体变量的定义、初始化和使用。

【例 7.1】给结构体变量赋值并输出各成员的值。注意观察对结构体变量的不同的赋值、输入和输出的方法。

参考代码：

```
#include<stdio.h>
#include<string.h>
int main(void)
{
    struct person
    {
        long num;
        char name[20];
        char sex;
        int age;
        float salary;
        char addr[20];
    }stu1, stu2;
    stu1.num=102;                              //直接赋值
    strcpy(stu1.name, "zhangsan");             //不能写成 stu1.name="zhangsan";
    stu1.age=23;
    strcpy(stu1.addr, "Qingdao");
    printf("Input sex and salary\n");
    scanf("%c%f", &stu1.sex, &stu1.salary);    //通过 scanf()函数动态赋值
    stu2=stu1;                                 //结构体变量整体赋值
    printf("Number=%ld\nName=%s\nAge=%d\nAddr=%s\n", stu2.num, stu2.name,
        stu2.age, stu2.addr);                  //只能分别输出结构体变量各成员变量的值
    printf("Sex=%c\nSalary=%f\n", stu2.sex, stu2.salary);
    return 0;
}
```

运行实例：

```
Input sex and salary
m
10000
Number=102
Name=zhangsan
Age=23
Addr=Qingdao
Sex=m
```

```
Salary=10000.000000
```

例 7.1 程序中用赋值语句给 num、name、age、addr 四个成员变量赋值，name 和 addr 是字符数组名，赋值时必须使用 strcpy()函数，不能使用赋值运算符直接赋值。用 scanf()函数动态地输入 sex 和 salary 成员值，然后把 stu1 的所有成员值整体赋值给 stu2，最后分别输出 stu2 的各个成员值。

【例 7.2】利用结构体类型，编程计算一名同学 5 门课的平均分。

【分析】本题主要练习结构体变量的初始化和结构体成员变量的引用方法。

参考代码：

```
#include<stdio.h>
int main(void)
{
    struct stuscore
    {
        char name[20];
        float score[5];
        float average;
    };
    struct stuscore x={"zhangsan", 90, 85, 70, 90, 98}; //定义结构体变量 x 并初始化
    int i;
    float sum=0;
    for(i=0; i<5; i++)
        sum+=x.score[i];                    //引用结构体成员变量
    x.average=sum/5;
    printf("The average score of %s is %4.1f\n", x.name, x.average);
    return 0;
}
```

运行实例：

```
The average score of zhangsan is 86.6
```

【练习 7.2】① 定义一个有关时间的结构体变量(其中包括小时、分钟、秒)，从键盘为其输入数据，并显示。

② 定义一个日期结构体变量(包括年、月、日)。计算该日在本年中是第几天？注意闰年问题。

7.2　结构体数组

结构体数组的每一个元素都是具有相同类型的结构体变量。在实际应用中，经常用结构体数组来表示具有相同数据结构的一个群体，如一个班的学生档案，一个车间职工的工资表等。

结构体数组的定义方法与结构体变量的定义类似，只需要说明它是数组类型。例如：

```
struct student
{
    int no;
    char name[20];
    char sex;
    float score;
}stu[5];
```

以上程序段定义了一个结构体数组 stu，它共有 5 个元素，分别是 stu[0]～stu[4]。每个数组元素都是 struct student 类型。对结构体数组可以进行初始化赋值。例如：

```
struct student
{
    int no;
    char name[20];
    char sex;
    float score;
}stu[5]={
        {101, "li ming", 'M', 90},
        {102, "zhang san", 'M', 90},
        {103, "he fang", 'F', 92.5},
        {104, "chen ling", 'F', 87},
        {105, "wang ming", 'M', 56}
       };
```

当对所有数组元素进行初始化赋值时，也可以省略数组长度。

【例 7.3】计算学生的平均成绩和不及格人数。

【分析】这个题目大家已经很熟悉了，在本例中只需要定义一个结构体类型，然后定义结构体数组，对结构体数组元素的成员进行相应运算。

参考代码：

```
#include<stdio.h>
struct student
{
    int no;
    char name[20];
    char sex;
    float score;
}stu[5]={
        {101, "li ming", 'M', 90},
        {102, "zhang san", 'M', 90},
        {103, "he fang", 'F', 92.5},
        {104, "chen ling", 'F', 87},
        {105, "wang ming", 'M', 56}
       };
```

```
int  main(void)
{
    int  i, c=0;
    float  ave, s=0;
    for(i=0; i<5; i++)
    {    s+=stu[i].score;
         if(stu[i].score<60)
              c+=1;
    }
    printf("s=%.2f\n", s);
    ave=s/5;
    printf("average=%.2f\ncount=%d\n", ave, c);
    return  0;
}
```

运行实例：

```
s=415.50
average=83.10
count=1
```

在调用函数传递参数时，用结构体变量的成员作参数，使用方法与普通变量一样。如果用结构体变量作实参，则所有成员按值传递给形参。如果用指向结构体变量(或数组)的指针作实参，则将结构体变量(或数组)的地址传给形参。

【例 7.4】下述程序用结构体变量作为函数参数。阅读程序，分析程序运行结果。

```
#include<string.h>
#include<stdio.h>
struct  STU
{    int  num;
     float  TotalScore;
};
void  f(struct  STU  p)
{    struct  STU  s[2]={{20044, 550}, {20045, 537}};
     p.num = s[1].num;
     p.TotalScore = s[1].TotalScore;
}
int  main(void)
{
     struct  STU  s[2]={{20041, 703}, {20042, 580}};
     f(s[0]);
     printf("%d    %3.0f\n", s[0].num, s[0].TotalScore);
     return  0;
}
```

上述程序的开始定义了一个结构体类型 STU，在 main()函数和 f()函数内分别定义了两个同名结构体数组，并赋了初值，调用函数时，使用了传值方式传递参数，f()函数对形参的改变不影响主函数的 s[0]，所以程序运行结果是：

20041　703

【练习 7.3】定义一个结构体数组用以保存 10 个学生的以下信息：学号、姓名、性别、家庭住址、6 门课程成绩。要求：

① 从键盘输入 10 个学生的数据。

② 显示每个学生 6 门课程中的最低分和最高分。

③ 显示 10 个学生中第一门课程的最低分和最高分。

④ 显示 10 个学生中有一门不及格和所有课程均不及格的人数。

⑤ 检索学号为 NUM 的学生的 6 门课程成绩，NUM 由键盘输入。

【练习 7.4】有以下程序：

```
struct STU
{   char name[10];
    int num;
};
void f1(struct STU c)
{   struct STU b={"LiSiGuo", 2042};
    c=b;
}
void f2(struct STU *c)
{   struct STU b={"SunDan", 2044};
    *c=b;
}
main()
{   struct STU a={"YangSan", 2041}, b={"WangYin", 2043};
    f1(a);   f2(&b);
    printf("%d %d", a.num, b.num);
}
```

运行程序后的输出结果是____。

A. 2041 2044　　B. 2041 2043　　C. 2042 2044　　D. 2042 2043

【练习 7.5】有如下定义：

```
struct person
{   char name[9];
    int age;
};
struct person class[10] = {"John", 17, "Paul", 19, "Mary", 18, "Adam", 16,};
```

根据上述定义，下述语句中能输出字母 M 的语句是_____。

A. printf("%c\n", class[3].name[0]);　　B. printf("%c\n", class[3].name[1]);

C. printf("%c\n", class[2].name[1]);　　D. printf("%c\n", class[2].name[0]);

7.3　线 性 链 表

练习 7.3 中学生的人数是固定的 10 人，因此在编程中可以定义一个结构体数组来保存学生的信息，数组长度为 10。但是如果将题目中的 10 人改为 n 个人，n 的值由用户输入得到，还能用结构体数组来描述学生信息吗？不能，因为 n 的值在程序运行时才能获得，也就是所需的内存空间的大小要在程序运行时才能确定，而数组内存空间的分配需要在程序编译时就确定。解决这个问题需要用到动态内存分配。

【例 7.5】输入一个整数 n，并且输入 n 个学生的信息(学号和姓名)，分配内存空间保存学生的信息，用完后释放内存空间。

【分析】本例中，n 的值及范围都未知，因此不能采用定义结构体数组的方式，可以采用动态内存分配的方式，一次性开辟 n 个学生信息的内存空间，此时的内存空间就是连续的，从本质上就相当于定义了一个数组，接着把输入的数据存入到该“数组”空间即可。

参考代码：

```
#include<stdio.h>
#include<stdlib.h>
int  main(void)
{
    struct  student
    {
        int  no;
        char  name[10];
    }*p;
    int  n, i;
    printf("请输入学生个数：");
    scanf("%d", &n);
    p=(struct  student*)malloc(n*sizeof(struct  student));
    /*开辟空间后，就相当于定义了一个名称为 p，长度为 n 的数组*/
    for(i=0; i<n; i++)
    {
        printf("请输入第%d 个学生的学号和姓名：\n", i+1);
        scanf("%d%s", &p[i].no, p[i].name);
    };
    for(i=0; i<n; i++)
        printf("Number=%d  Name=%s\n", p[i].no, p[i].name);
    free(p);                    /*释放内存空间*/
    return  0;
}
```

运行实例：

```
请输入学生个数：2
请输入第 1 个学生的学号和姓名：
1 zhangsan
请输入第 2 个学生的学号和姓名：
2 lisi
Number=1  Name=zhangsan
Number=2  Name=lisi
```

【程序小结】例 7.5 中定义了结构体类型 struct student 和 struct student 类型的指针变量 p，然后动态分配 n 个 sizeof(struct student)字节大小的内存空间，并让 p 指向这段空间的首地址，本质上和以 p 为数组名，长度为 n 的结构体数组一样，此时该空间内的值为随机值，接下来通过 scanf()函数给这段空间赋初值，即给以 p 所指向的各成员赋值，然后用 printf()函数输出该段内存空间各个元素的各成员的值。最后用 free()函数释放内存空间。

例 7.5 程序中包含申请内存空间，使用内存空间，释放内存空间三部分，完成了对内存空间的动态使用。

7.3.1 链表的概念

例 7.5 中输入的学生个数为 n 个，采用动态内存分配函数一次开辟了 n 个学生信息的内存空间，此时开辟的内存空间都是连续的，从本质上相当于定义了一个长度为 n 的数组，但如果输入学生信息的个数未知，对某个班级的学生信息进行处理，以上方法将无法解决问题。

此时只能输入一个学生的信息，为这个学生信息开辟一段内存空间，这样不同学生信息的内存空间将是随机的，不一定连续，因此要通过一个学生信息找到下一个学生的学生信息，就必须在学生信息中保存下一个学生信息的存放地址，因此应该在描述学生信息的结构体中增加一个成员变量，变量的类型为指针类型，而指针的基类型只能是描述学生信息的结构体类型。

如果我们把描述每个学生信息的结构体称为一个结点，那么通过一个结点可以找到下一个结点，这样的“环环相扣”就好像一条链子，因此我们将这种数据结构称为线性链表，线性链表分为单向链表和双向链表，我们重点介绍单向链表，以下简称其为链表。

链表是一种动态分配存储单元的存储结构。链表由若干个结点构成，就好像数组中的数组元素，每个结点具有相同的数据类型，并且每个结点中都有一个可以指向其他结点的指针。因此线性链表中的数据元素在内存中不需要连续存放，而是通过指针将各结点链接起来，就像一条“链子”一样。

根据以上的分析，链表中的结点应定义为结构体类型，而且该类型中应该有一个指向下一个结点的指针变量，称为指针域，其余的成员变量称为数据域，用来描述其他信息，如学生信息中的学号、姓名等。下面以学生成绩链表结点为例，定义结点结构体如下：

```
struct  student
{
    int  no;
    float  score;
    struct  student  *next;
};
```

这样，使用结点结构体就可以定义多个结构体变量(结点)，而且第一个结点的指针域存储第二个结点的首地址，第二个结点的指针域存储第三个结点的首地址……最后一个结点不指向任何一个结点，因此它的指针域可以置为空(NULL)，如图 7.3 所示。

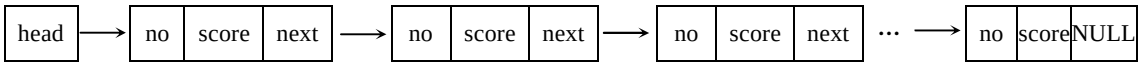


图 7.3　链表的基本结构

使用链表，就可以解决上述问题，把每个学生的信息定义为一个 struct student 类型的结点，可以采用动态分配内存的方式为其分配内存空间。班级里有多少个学生就建立多少个结点，若学生退学或转学，则可以删除结点，释放内存空间，从而实现对内存空间的有效利用。同时，由于结点内存单元的申请与释放，各结点间的地址将变得不连续，但这不会影响对所有学生信息的处理，因为每个结点都通过指针域进行连接。这就是链表的作用。

7.3.2　链表的基本操作

对链表的基本操作主要有链表的建立、查找、插入、删除、输出等。其中链表的建立从某种程度上来说就是对链表进行插入的过程，而链表的查找和输出都是对链表进行遍历的过程。下面定义一个简单的学生成绩结点，并据此进行详细介绍。

```
struct linknode
{
    int no;
    float score;
    struct linknode *next;
};
```

1. 建立链表

从图 7.3 中可以看出，链表中的元素在内存中可以不连续，所以要找某一元素，必须先找到其上一元素，根据它提供的地址才能找到所需要的元素。如果不提供“头指针”(head)，则整个链表都无法访问。链表如同一条链子一样，环环相扣，中间不能断开。打个通俗的比方，幼儿园老师带领孩子出来散步，老师牵着第 1 个小孩的手，第 1 个小孩的另一只手牵着第 2 个小孩，依次下去，就形成了一条“链”，最后一个孩子有一只手空着，即为“链尾”。要找到这个队伍，必须先找到老师，然后顺序找每一个孩子。

建立链表的步骤如下。

① 定义变量 head、tail、pnew。

head——指针类型，指向链表的第一个结点，头结点。

tail——指针类型，指向链表的最后一个结点，尾结点。

pnew——指针类型，代表新申请的结点，也就是待插入到链表中的结点。

② 链表的建立是一个从无到有的过程。因此最初有：

```
head=NULL;
tail=NULL;
```

③ 新申请结点：

首先为新结点动态分配内存空间，并让 pnew 指向新申请内存空间的首地址，具体如下：

```
pnew=(struct linknode*)malloc(sizeof(struct linknode));
```

其次为新申请结点设置数据域，可以通过直接赋初值的方式也可以通过输入的方式，如：

pnew->no=1;　或者　scanf("%d", &pnew->no);

④ 设置新申请结点的指针域：作为新生成结点，它没有下一个结点，因此指针域为NULL：

pnew->next=NULL;

⑤ 将新申请的结点插入链表：

将新申请的结点插到链表中的方法是将新结点插到链表的最后。需要考虑两种情况：即空链表和非空链表两种情况。

如果链表为空，那么头结点 head 为新申请的结点 pnew，head=pnew，否则应将 pnew 插入到尾结点 tail 之后，tail->next= pnew，无论是哪一种情况，因为新申请的结点要插入到链表尾部，所以尾结点 tail 都应该修改为 pnew，tail= pnew。

N-S 流程图如图 7.4 所示。

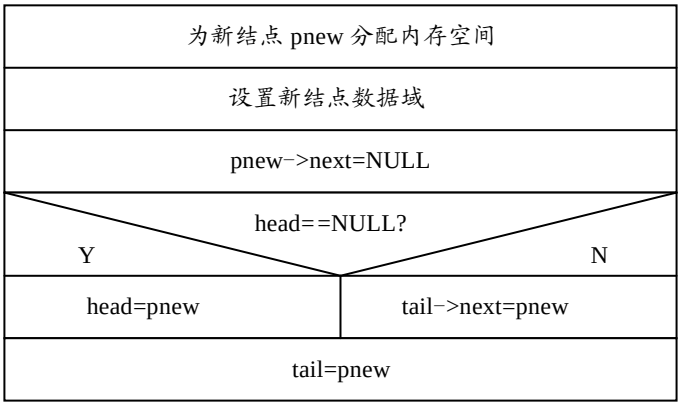


图 7.4　建立链表的 N-S 图

【例 7.6】编写创建链表的 Create()函数。

参考代码：

```
struct linknode *Create()
{
    struct linknode *head=NULL, *tail=NULL, *pnew;
    int no;
    float score;
    while(1)                                    // 创建线性链表
    {
        printf("Input the new no and score:\n");
        scanf("%d%f", &no, &score);             // 输入学生学号和成绩
        if(no==0)                               // 数据为 0 时退出循环
            break;
        // 创建一新结点，分配内存空间
        pnew=(struct linknode *)malloc(sizeof(struct linknode));
        if(pnew==NULL)                          // 分配内存空间失败
        {
```

```
            printf("No enough memory!\n");
            return(NULL);
        }
        pnew->no=no;                    // 为新结点数据域赋值
        pnew->score=score;
        pnew->next=NULL;                // 为新结点指针域赋值
        if(head==NULL)                  // 如果原链表为空
            head=pnew;                  // 新结点就是头结点
        else
            tail->next=pnew;            // 将新结点插入链表尾部
        tail=pnew;                      // 新结点是新的尾结点
    }
    return head;
}
```

2. 查找结点

查找是按照给定条件对链表进行遍历，必须从头结点 head 开始查找。如果找到，则返回找到结点的地址，查找成功；否则查找失败，返回 NULL。

链表作为一种物理存储单元上非连续、非顺序的存储结构，不能像数组一样利用下标进行遍历，但链表的指针域包含了后继结点的存储地址，可依此对链表的结点进行遍历。

【例 7.7】编写一个 Search()函数，实现按照结点在链表中的位置即索引进行查找，如查找链表中第 3 个结点的学生信息。

参考代码：

```
struct linknode *Search(struct linknode *head, int n)
{
    int i=1;
    struct linknode *p=head;
    for(p=head; p!=NULL; p=p->next, i++)
        if(i==n)
            break;
    if(i<n)
        return NULL;
    else
        retrun p;
}
```

【总结】从链表的查找可以看出链表的一个缺点，对链表不能像数组那样实现数据的随机存取，必须从头结点开始进行遍历。

【思考】如果将查找条件改为按学号进行查找，应如何修改代码？

3. 插入结点

首先找到要插入结点的位置，这对应于链表的查找过程。

对链表进行插入，应遵循先连后断的原则。

例如在图 7.5 中，要在 p 和 q 两个结点之间插入 r 结点，先将 r 结点的 next 域指向 q 结

点，然后将 p 结点的 next 域指向 r 结点，断开原来 p 和 q 结点间的连线。这样通过 p 就可以找到 r 结点，通过 r 结点又可以找到 q 结点，一个包含结点 r 的新链表就建立起来了。

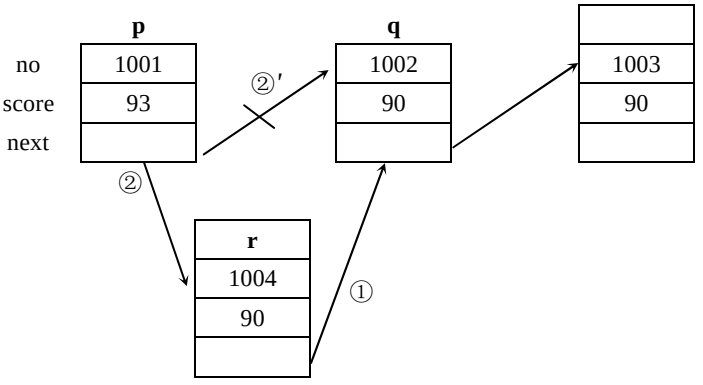


图 7.5　链表的插入

【例 7.8】编写在链表中完成插入结点操作的 insert 函数，在结点 p 之后插入结点 r。

参考代码：

```
void insert(struct linknode *p, struct linknode *r)
{   r->next=p->next;
    p->next=r;
}
```

【思考】如果要将结点插入到头结点之前，应如何操作？

4. 删除结点

原链表如图 7.6 所示，现在想删除结点 p 的后继结点 q。

基本思想：只要让结点 p 指向结点 q 的后继结点，然后释放结点 p 的后继结点的空间。因此应将删除前 p 的后继结点提前保存下来。删除后的链表如图 7.7 所示。

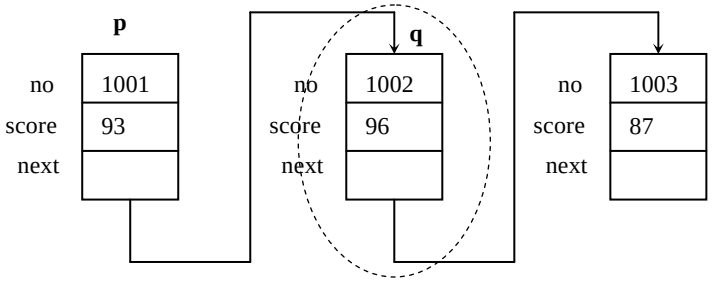


图 7.6　删除结点 q 前的链表

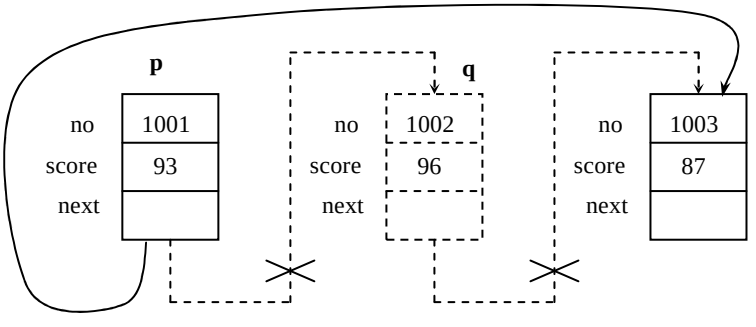


图 7.7　删除结点 q 后的链表

【注意】在链表中删除结点应遵循先接后删的原则。

【例 7.9】编写在链表中完成删除结点操作的 Delete()函数(删除结点 p 的后继结点)。

参考代码：

```
void Delete(struct linknode *p)
{
    struct linknode *q;
    q=p->next;                          // q 指向要删除的结点
    p->next=q->next;                    // 删除结点 q
    free(q);                            // 释放被删除结点的内存单元
}
```

【思考】如果待删除的结点是链表的头结点该如何操作？

5. 链表的输出

进行输出操作时，仍要对链表中的结点进行遍历，只不过在遍历的过程中进行将数据域的值显示出来的操作。

基本思想：使指针 p 指向单链表的头指针 head，输出其数据值，接着通过 p 的指针域 next 获得下一个结点的地址，让 p 指向下一个结点，再输出其数据值，如此进行下去，直到输出尾结点的数据项为止，即 p 为 NULL 为止。

【例 7.10】编写完成输出链表操作的 Display()函数。

参考代码：

```
void Display(struct linknode *head)
{
    struct linknode *p;
    for(p=head; p!=NULL; p=p->next)
    printf("Num：%d, Score：%f\n", p->no, p->score);
}
```

【练习 7.6】有以下结构体说明和变量的定义，且指针 p 指向变量 a，指针 q 指向变量 b：

```
struct node
{   char data;
    struct node *next;
}a, b, *p=&a, *q=&b;
```

下述语句中，不能把结点 b 连接到结点 a 之后的语句是_____。

A. p=&a;　　B. p->next=&b;　　C. (*p).next=q;　　D. p->next=q;

【练习7.7】有以下结构体说明和变量定义：

```
struct node
{int data;    struct node *next;}*p, *q, *r;
```

如图7.8所示，指针p，q，r分别指向链表中的三个连续结点。

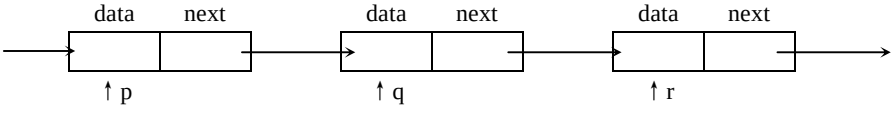


图7.8　一个链表的示意图

现要将q所指结点从链表中删除，同时要保持链表的连续，以下语句中不能完成这项指定操作的语句是____。

A. p->next=q->next;　　B. p->next=p->next->next;

C. p->next=r;　　D. p=q->next;

7.4 共 用 体

共用体类型也是一种构造数据类型，在共用体变量中，可以把几种不同类型的数据存放于同一段内存中。这种使几个不同变量占用同一段内存空间的结构称为共用体，又称为共同体。对共用体后面赋值的成员变量将覆盖前面的成员变量，因此各成员变量不能同时赋值。

7.4.1 共用体类型定义

共用体类型的定义跟结构体类型的定义格式基本相同，仅仅是关键字不同。结构体的关键字是 struct，共用体是 union。共用体类型定义的一般格式如下：

```
union 共用体名
{
    数据类型名1 共用体成员变量名1;
    数据类型名2 共用体成员变量名2;
    ……
    数据类型名n 共用体成员变量名n;
};
```

例如：

```
union date
{
    int i;
    char ch;
    double f;
};
```

上述共用体类型 date 包含 3 个成员变量。它们共用同一地址的内存单元，如图 7.9 所示。共用体所占内存的大小是成员中占内存最大的成员的大小。date 的成员中，f 所占内存最大，占 8 个字节的内存空间，因此，date 大小也是 8 个字节。

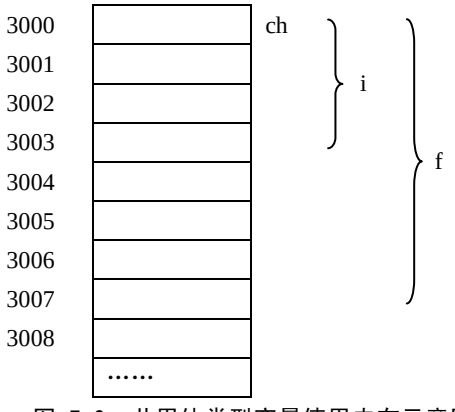


图 7.9　共用体类型变量使用内存示意图

结构体类型中的每一个成员变量都占用独立的内存空间，而共用体类型中的成员变量则共享同一段内存单元。如果前面定义的共用体类型定义成结构体类型，则所占的存储空间为：

sizeof(int)+sizeof(char)+sizeof(double)=4+1+8=13 字节

7.4.2　共用体变量的定义、引用

1. 共用体变量的定义

与结构体变量的定义方法类似，共用体变量定义也有 3 种方法。下面通过实例说明共用体变量的定义方法。

① 先定义共用体类型，再定义共用体变量：

```
union date
{
    int i;
    char ch;
    double f;
};
union date x, y[10];
```

② 在定义共用体类型的同时，定义共用体变量：

```
union date
{
    int i;
    char ch;
    double f ;
}x, y[10];
```

③ 省略共用体名，定义共用体变量：

```
union
{
    int i;
    char ch;
    double f ;
}x, y[10];
```

2. 共用体变量的引用

共用体变量的引用格式与结构体变量的引用格式相同，如果通过共用体变量来引用成员变量，则用“.”运算符，如果通过共用体指针来引用成员变量，则用“->”运算符。例如有下述定义语句：

```
union date x, y[10], *p;
*p=&x;
```

则其引用其成员变量的方式为 x.i，x.ch，x.f，y[0].i，y[0].ch，y[0].f，p->i，p->ch，p->f 等。

7.4.3　共用体变量的赋值

共用体变量的赋值可以分为在定义时赋初值，以及定义后在程序中赋值两种。

1. 定义时给共用体变量赋初值

在定义共用体变量时赋初值，只能给其第一个成员赋初值，不能像结构体一样给所有成员赋初值。例如：

```
union date x={20};      //把 20 赋给了成员 i
union date x={'A'};     //把'A'赋值给成员 i，即 i 的值为 65('A'的 ASCII 码)
union date x={20, 'B', 6.5};      //这种赋值是错误的，{}中只能有一个值
union date x=20;                  //错误，初值必须用“{”和“}”括起来
```

2. 在程序中给共用体变量赋值

共用体变量定义以后，如果要对其赋值，只能对其成员赋值，不可对其整体赋值。例如：

```
union date x, y[10], *p;
x.i=10;
p=&x;
p->f=6.5;
y[0].i='A';
```

与结构体变量一样，相同类型的共用体变量之间可以相互赋值。例如：

```
union date x={20}, y;
y=x;
```

【注意】① 由于共用体变量所有成员共用同一段内存空间，所以只有最后一次赋值的共用体成员的值有效，之前赋值的成员值将被覆盖。例如：

```
union date x;
x.i=10;    x.ch='A';    x.f=6.5;
```

执行上述语句后，只有 x.f 的值是有效的，x.i 和 x.ch 无意义了。

② 由于共用体变量所有成员共用同一段内存空间，所以共用体变量与其各成员的地址是相同的，即&x.i，&x.ch，&x.f，&x 的地址是相同的。

【练习 7.8】有以下程序：

```
union myun
{
    struct
    {int x, y, z;}u;
    int k;
}a;
void main()
{
   a.u.x=4;
   a.u.y=5;
   a.u.z=6;
   a.k=0;
   printf("%d\n", a.u.x);
}
```

运行程序后的输出结果是_____。

A. 4　　B. 5　　C. 6　　D. 0

【练习7.9】若有以下定义和语句：

```
union data
{   int i;
```

```
    char  c;
    float  f;
}x;
int  y;
```

以下语句中正确的语句是_____。

A. x=10.5;　　B. x.c=101;　　C. y=x;　　D. printf("%d\n", x);

7.5　枚 举 类 型

在实际应用中，有的变量只有几种可能取值。如人的性别只有两种可能取值，星期几只有七种可能取值。在 C 语言中可以把这类变量定义为枚举类型。

所谓枚举是指将变量的值一一列举出来，变量只限于在列举出来的值的范围内取值。

1. 枚举类型的定义

枚举类型定义的一般形式为：

```
enum  枚举名{枚举值表};
```

在枚举值表中应罗列出所有可用值。这些值也称为枚举元素。例如：

```
enum  weekday{sun, mon, tue, wed, thu, fri, sat};
```

该枚举名为 weekday，共有 7 个枚举值，即一周中的七天。凡被说明为“enum weekday”类型的变量，其取值只能是七天中的某一天。

2. 枚举变量的定义

如同结构体和共用体一样，枚举变量也可用不同的方式定义，即先定义类型后定义变量，同时定义类型和变量或直接定义变量。

可采用下述三种方式把变量 a，b，c 说明为上述的枚举类型：

①
```
enum  weekday{sun, mou, tue, wed, thu, fri, sat};
enum  weekday a, b, c;
```

②
```
enum  weekday{sun, mou, tue, wed, thu, fri, sat}a, b, c;
```

③
```
enum{sun, mou, tue, wed, thu, fri, sat}a, b, c;
```

其中，sun，mon，…，sat 等称为枚举元素或枚举常量，它们是用户定义的标识符。

枚举类型在使用中有以下规定：

① 枚举值是常量，不是变量。不能在程序中用赋值语句再对它赋值。

例如下述对枚举元素再进行赋值都是错误的：

```
sun=5;
mon=2;
sun=mon;
```

② 枚举元素本身由系统定义了一个表示序号的数值，从 0 开始顺序定义为 0，1，2，…。如在 weekday 中，sun 值为 0，mon 值为 1，…，sat 值为 6。

如果在定义枚举类型时指定元素的值，也可以改变枚举元素的值。例如：

```
enum  weekday{sun=7, mon=1, tue, wed, thu, fri, sat}day;
```

这时，sun 为 7，mon 为 1，以后元素顺次加 1，所以 sat 就是 6 了。

③ 枚举值可以用来做判断。例如：

```
if(day==mon){…}
if(day>mon){…}
```

枚举值的比较规则是：按其在说明时的顺序号比较，如果说明时没有人为指定，则第一个枚举元素的值为 0。例如，mon>sun，sat>fri。

④ 一个整数不能直接赋给一个枚举变量，必须进行强制类型转换后才能赋值。例如：

```
day=(enum  weekday)2;
```

这个赋值语句将顺序号为 2 的枚举元素赋给 day，相当于语句“day=tue;”。

【例 7.11】枚举类型变量应用示例。

```
#include<stdio.h>
void  main()
{
    enum  weekday{sun, mon, tue, wed, thu, fri, sat }a, b, c;
    a=sun;
    b=mon;
    c=tue;
    printf("%d, %d, %d", a, b, c);
}
```

运行实例：

```
0, 1, 2
```

【说明】① 只能把枚举值赋给枚举变量，不能把元素的数值直接赋予枚举变量。例如：

```
a=sun;
b=mon;
```

是正确的。

而

```
a=0;
b=1;
```

是错误的。

② 如一定要把数值赋予枚举变量，则必须用强制类型转换。例如：

```
a=(enum  weekday)2;
```

其意义是将顺序号为 2 的枚举元素赋予枚举变量 a，相当于“a=tue;”。

③ 枚举元素不是字符常量也不是字符串常量，使用时不要加单、双引号。

【例 7.12】从键盘输入一个整数，显示该整数对应的星期几英文名称。如 1 对应 Monday，0 对应 Sunday。

```
#include<stdio.h>
void  main()
{
    enum  weekday{sun, mon, tue, wed, thu, fri, sat}day;
    int  k;
```

```
    printf("Input a number(0～6)：");
    scanf("%d", &k);
    day=(enum weekday)k;
    switch(day)
    {
      case sun:    printf("sunday\n");   break;
      case mon:    printf("monday\n");   break;
      case tue:    printf("tuesday\n");  break;
      case wed:    printf("wednesday\n"); break;
      case thu:    printf("thursday\n");  break;
      case fri:    printf("friday\n");   break;
      case sat:    printf("satday\n");   break;
      default:     printf("input error\n"); break;
    }
}
```

运行实例：

```
Input a number(0～6)：1
monday
```

在例 7.12 的程序中，枚举常量与枚举变量可以进行比较，但要输出枚举常量对应的英文单词，不能使用以下语句：

```
printf("%s", mon);
```

因为枚举常量 mon 为整数值，而非字符串。

在使用枚举变量时，主要关心的不是它的值的大小，而是其表示的状态。

【练习 7.10】已知枚举类型定义为“enum color{yellow, red, blue, green, white, black};”从键盘输入一个整数(0～5)，显示与其相对应的枚举常量的英文名称。

7.6　自定义类型名

C 语言允许给已有的数据类型起一个别名，这些已有的数据类型包括基本数据类型，如 int，float，double，char 等，也包括构造数据类型，如数组、指针、结构体等，声明的方式如下：

```
typedef 已有类型名 别名;
```

所谓“已有类型名”就是已经声明的合法数据类型，而“别名”只要是合法的 C 语言标识符即可。例如：

```
typedef int INTEGER;
typedef struct
{
```

```
    int  year, month, day;
}DATE;
```

则 INTEGER 和 DATA 都是新的数据类型名，可以用来定义变量。例如：

```
INTEGER  i;
DATE  date;
```

【说明】① typedef 语句并未产生新的数据类型，只是为已有的数据类型起了个别名。

② 为了区分起见，别名一般采用大写字母表示。

③ 注意区分以下两段代码的不同。

```
typedef  struct
{
    int  year, month, day;
}DATE;
```

```
struct
{
    int  year, month, day;
}date;
```

其中 DATE 是一种结构数据类型，date 是一个该结构类型的变量。

【练习 7.11】以下各选项中，不能把 c1 定义成结构体变量的是_____。

A.
```
typedef  struct
{   int  red;
    int  green;
    int  blue;
}COLOR;
COLOR  c1;
```

B.
```
struct color c1
{   int  red;
    int  green;
    int  blue;
};
```

C.
```
struct  color
{   int  red;
    int  green;
    int  blue;
}c1;
```

D.
```
struct
{   int  red;
    int  green;
    int  blue;
}c1;
```

7.7 编译预处理

7.7.1 编译预处理命令简介

编译预处理功能是 C 语言的一个重要特征。一个用高级语言编写的源程序代码在计算机上运行，必须先通过编译系统将其翻译为目标代码。编译包括词法分析、语法分析、代码生成、代码优化等步骤，有时在编译之前还要做某些预处理工作，如去掉注释，变换格式等。

C 语言源程序中以#开头、以换行符结尾的行称为编译预处理指令。编译预处理指令不是 C 语言的语法成分，而是传给编译程序的各种指令。C 语言的编译预处理命令包括：

① 宏定义：

```
#define
#undef
```

② 文件包含：

```
#include
```

③ 条件编译：

#if

#ifdef

#else

#elif

#endif

④ 其他：

#line

#error

#pragma

本节主要介绍前三种预处理命令的用法。

7.7.2　宏定义

宏定义分为两种：不带参数的宏定义和带参数的宏定义。

1. 不带参数的宏定义

#define 指令定义一个标识符来代表一个字符串，在源程序中发现该标识符时，都用该字符串替换，以形成新的源程序。这种标识符称为宏名(macro name)，将程序中出现的与宏名相同的标识符替换为字符串的过程称为宏替换(macro substitution)。宏替换的操作是在预编译时进行的。

不带参数的宏定义的一般格式如下：

#define 标识符 字符串

其中：

① #define 是宏定义的指令名称。

② 标识符就是宏名，它被定义代表后面的字符串。

③ 字符串是宏的内容文本，也称为宏体，可以是任意以回车换行符结尾的文字。

④ 宏一般不能用分号结尾，除非程序员故意这样做。

【例 7.13】分析下列两个程序段。

```
#define PI 3.1415
#include<stdio.h>
void main()
{
    double r, s;
    scanf("%lf", &r);
    s=PI*r*r;
    printf("s=%f\n", s);
}
```

```
#include<stdio.h>
void main()
{
    double r, s;
    scanf("%lf", &r);
    s=3.1415*r*r;
    printf("s=%f\n", s);
}
```

对于宏定义应注意以下几点：

① 宏替换时仅仅将源程序中与宏名相同的标识符替换成宏的内容文本，并不对宏的内容文本做任何处理。

② C 语言程序员通常用大写字母来定义宏名，以便与变量名相区别。这种做法可以帮助

读者迅速识别发生宏替换的位置。同时最好把所有宏定义放在文件的最前面或另一个单独的文件中，不要把宏定义分散在文件的多个位置。

③ 宏定义时，如果字符串太长，需要写多行，可以在行尾使用反斜线“\”续行符。例如：

```
#define LONG_STRING "this is a very long string that is\
used as an example"
```

注意双引号包括在替代的内容之内。

④ 宏名的作用域是从#define 定义之后直到该宏定义所在文件结束，但通常把#define 宏定义放在源程序文件的开头部分。如果需要终止宏的作用域，可以使用#undef 命令，其一般格式是:

```
#undef 标识符
```

【例 7.14】下面的程序中宏 N 的作用域在第 1 行～第 11 行之间。

```
1   #define N 100              //宏定义
2
3   int sum()
4   {
5       int i, s=0;
6       for(i=1; i<=N; i++)
7           s+=i;
8       return(s);
9   }
10
11  #undef N                   //宏取消
12
13  void main()
14  {
15      int a;
16      a=sum();
17      printf("%d\n", a);
18  }
```

N 的作用域(有效范围)

⑤ 宏定义可以嵌套，但不能进行递归定义。例如下列嵌套定义是正确的：

```
#define R 2.0
#define PI 3.14159
#define L 2*PI*R
#define S PI*R*R
```

在编译预处理时，宏 L 被 2*3.14159*2.0 替换，宏 S 被 3.14159*2.0*2.0 替换。但下面的宏定义是错误的：

```
#define M M+10      //不可递归定义宏
```

⑥ 程序中字符串常量即双引号中的字符，不作为宏进行宏替换操作。例如：

```
#define XYZ this is a test
```

```
printf("XYZ");
```

此时输出的将是“XYZ”，而不是“this is a test”。

⑦ 宏定义一般以换行结束，不要用分号结束，以免引起不必要的错误。例如：

```
#define PI 3.14159;
```

则对于“a=PI*2*2;”经替换后变成了“a=3.14159;*2*2;”，一看就知道是错误的。

⑧ 关于重复宏定义：重复的宏定义以最后一次定义为准。尽量不要重复定义宏，那样会让程序的可读性变差，并且容易出错，难以维护。

【例 7.15】重复宏定义示例。

```
#define PI 3.14
#define PI 10.48
```

这样定义后，此后程序中 PI 的值是 10.48，因为这是最后的定义。

⑨ 在定义宏时，如果宏是一个表达式，要将这个表达式用()括起来，否则可能引起非预期的结果。

【例 7.16】下面左边的程序在定义宏时，没有将宏表达式括起来，观察运行效果。

源　程　序	宏替换后的程序

#define X 10
#define Y 20
#define Z X+Y
#include<stdio.h>
void main()
{
　int a=2, b=3;
　a*=Z;
　b=b*Z;
　printf("a=%d, b=%d\n", a, b);
	#include<stdio.h>
	void main()
	{
	　int a=2, b=3;
	　a*=10+20;
	　b=b*10+20;
	　printf("a=%d, b=%d\n", a, b);
	}

运行实例：

```
a=60, b=50
```

宏替换时，语句“b=b*Z;”没有替换成“b=b*(10+20);”，而是替换成“b=b*10+20;”。如果希望把“b=b*Z;”替换成“b=b*(10+20);”，应该将 Z 的定义改成：

```
#define Z (X+Y)
```

2. 带参数的宏定义

带参数的宏定义的一般形式是：

```
#define 标识符(参数列表) 字符串
```

其中：① 参数表由一个或多个参数构成，参数只有参数名，没有数据类型符，参数之间用逗号隔开，参数名必须是合法的标识符。

② 字符串是宏的内容文本，也称为宏体，其中通常会引用宏的参数。

预编译器按下述步骤处理带参数的宏：首先将宏内容文本中的宏参数替换成实参文本，这样形成了宏的实际内容文本，再将这个宏的实际内容文本替换源程序中的宏标识符。

【例 7.17】带参数的宏定义与宏替换实例。

源　程　序	宏替换后的程序

#define SQRT(x) ((x)*(x))
#define MAX(x, y) (((x)>(y))?(x):(y))
#include<stdio.h>
void main()
{ float a=-2.5, b=-3.2;
a=MAX(a, b)+3;
printf("sqrt=%f\n", SQRT(a));
}
	#include<stdio.h>
	void main()
	{
	float a=-2.5, b=-3.2;
	a=(((a)>(b))?(a):(b))+3;
	printf("sqrt=%f\n",(a)*(a));
	}

预编译器在处理上面源程序中的 MAX(a, b)时，首先将 MAX(x, y)宏内容文本中的 x 替换成 a，将 y 替换成 b，形成新的宏内容是“((a)>(b)?(a):(b))”，然后将 MAX(a, b)替换成“((a)>(b)?(a):(b))”。在处理 SQRT(a)时，首先将 SQRT(x)宏内容文本中的 x 替换成 a，形成新的宏内容是“((a)*(a))”，然后将 SQRT(a)替换成“((a)*(a))”。

由于使用带参数的宏时，参数大多是表达式，宏内容本身也是表达式，因此，不但需要将整个宏内容括起来，而且还要将宏参数用“(”和“)”括起来，否则可能引起非预期的结果。例如下述宏定义：

```
#define SQRT(x) (x*x)
```

如果程序中有某语句为“a= SQRT(2+3);”，则预编译后该语句是“a=(2+3*2+3);”，a 的值将是 11，而不是我们所希望的 25。将 SQRT 宏的定义改成下面的形式：

```
#define SQRT(x) ((x)*(x))
```

就可以避免上述的错误。

宏定义语句一般位于函数外部，必须单独占一行，程序中定义宏时，必须在#define 命令的最后按回车键，否则会引起编译错误。

【注意】定义带参数的宏时，宏名与包含参数的圆括号之间不能有空白符，否则就会变成定义一个不带参数的宏，而要替换的内容是空白符之后的所有字符，从而出错。例如：

```
#define S (r) PI*r*r
```

相当于定义了不带参宏 S，S 代表字符串“(r) PI*r*r”。

【练习7.12】有以下程序：

```
#include<stdio.h>
#define SUB(a) (a)-(a)
main()
{ int a=2, b=3, c=5, d;
  d=SUB(a+b)*c;
  printf("%d\n", d);
}
```

运行程序后的输出结果是_____。

A. 0　　B. -12　　C. -20　　D. 10

【练习 7.13】有以下程序：

```
#define f(x) x*x
main()
 { int i;
```

```
    i=f(4+4)/f(2+2);
    printf("%d", i);
}
```

运行程序后的输出结果是_____。

A. 28　　B. 22　　C. 16　　D. 4

7.7.3　文件包含

文件包含是指一个 C 源程序通过#include 命令将另一个文件(通常是.c，.cpp 或.h 文件)的全部内容包含进来。文件包含处理命令的一般格式为：

#include<包含文件名>　　或　　#include"包含文件名"

预编译器是这样来处理#include 命令的：将包含文件的内容插入到源程序中#include 命令的位置，以形成新的源程序。如图 7.10 所示。

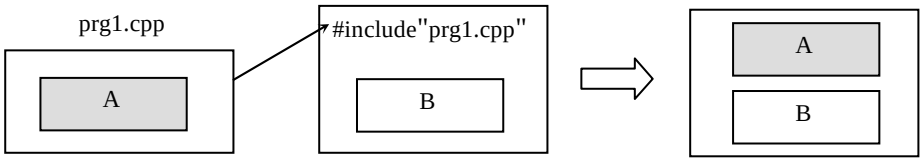


图 7.10　文件包含**#include** 命令预编译示意图

【例 7.18】文件包含实例。如果有文件 head.h 和 func.c，它们的内容如下：

head.h

```
#include<stdio.h>
#define  NUM  10
```

func.cpp

```
int  max(int  x, int  y)
{    return (x>y ? x : y);
}
int  getnum()
{   int  a;
    scanf("%d", &a);
    return  a;
}
```

下表列出了包含 head.h 和 func.cpp 的 prg.c 源程序文件和经预处理后的新程序。

源　程　序	预处理后的新程序

#include"head.h"
#include"func.c"
void main()
{
　　int a, b, c;
　　a=getnum();
　　b=getnum();
　　c=max(max(a, b), NUM);
	#include<stdio.h>
	#define NUM 10
	int max(int x, int y)
	{ return (x>y ? x : y);
	}
	int getnum()
	{ int a;
	　　scanf("%d", &a);

源　程　序	预处理后的新程序

printf("MAX=%d\n", c);
}
	return a;
	}
	void main()
	{
	int a, b, c;
	a=getnum();
	b=getnum();
	c=max(max(a,b), NUM);
	printf("MAX=%d\n", c);
	}

1. 使用文件包含的优点

一个大程序，通常分为多个模块，并由多个程序员分别编程。有了文件包含功能，就可以将多个模块公用的数据(如符号常量和数据结构)或函数,集中到一个单独的文件中(如上例中的文件 head.h 和 func.c)。这样，凡是要使用其中数据或调用其中函数的程序员，只要使用文件包含功能，将所需文件包含进来即可，不必再重复定义它们，从而减少重复劳动。

2. 两种文件包含格式的区别

① 使用一对尖括号<和 >：直接到系统指定的“文件包含目录”查找被包含的文件。在 VC 下，可以激活 Tools 菜单，选择“Options…”命令，在多页框中选择 Directories，再在“Show directories for:”组合框中选择“include files”，就可以在 Directories 文本框中看到 VC 系统的文件包含目录。

② 使用一对双引号“"”：系统首先到当前目录下查找被包含的文件，如果没找到，再到系统指定的“文件包含目录”中去查找。一般情况下，使用双引号比较保险。注意“"”之间可以指定包含文件的路径。如：

```
#include"c:\\prg\\p1.h"
```

表示将把 C 盘 prg 目录下的 p1.h 文件的内容插入到此处(字符串中要表示“\”,必须使用转义字符“\”，因此要表示目录分隔符需要用符号“\\”)。

【说明】① 常用在文件头部的被包含文件，称为“标题文件”或“头部文件”，这类文件常以“.h”(head)作为后缀，简称为头文件。在头文件中，除了可以包含宏定义外，还可以包含外部变量定义、结构类型定义等。

② 一条包含命令只能指定一个被包含文件。如果要包含 n 个文件,则要用 n 条包含命令。

③ 文件包含可以嵌套，即被包含文件中又包含另一个文件。

7.7.4 条件编译

一般情况下，源程序中所有的语句都参加编译，但有时也希望根据一定的条件编译源文件的不同部分，这就是条件编译。条件编译使得同一源程序在不同的编译条件下得到不同的目标代码。

条件编译有几种常用的形式，现分别进行介绍。

1. **#if～#endif 形式**

#if～#endif 形式的条件编译的格式如下：

```
#if 条件 1
    程序段 1
```

```
#elif 条件 2
    程序段 2
……
#else
    程序段 n
#endif
```

作用：如果条件 1 为真就编译程序段 1，否则如果条件 2 为真就编译程序段 2……如果各条件都不为真就编译程序段 n。

【说明】① elif 是 else if 的缩写，但是不可以写成 else if。

② 可以没有#elif 和#else，但必须有#endif，它是#if 命令的结尾符。

③ #elif 命令可以有多个。

④ if 后面的条件必须是一个常量表达式，通常会用到宏名，条件可以不加括号“()”。

⑤ 每个命令独占一行。

【例 7.19】利用 ACTIVE_COUNTRY 定义货币的名称程序示例。

源　程　序	预处理后的新程序

#define USA 0
#define ENGLAND 1
#define FRANCE 2
#define ACTIVE_COUNTRY USA
#if ACTIVE_COUNTRY==USA
　char *currency="dollar"; //有效
#elif ACTIVE_COUNTRY==ENGLAND
　char *currency="pound";
#else
　char *currency="france";
#endif
#include<stdio.h>
void main()
{
　float price1,price2,sumprice;
　scanf("%f%f", &price1, &price2);
　sumprice=price1+price2;
　printf("sum=%.2f%s",sumprice,currency);
}
	char *currency="dollar";
	#include<stdio.h>
	void main()
	{
	　float price1, price2, sumprice;
	　scanf("%f%f", &price1, &price2);
	　sumprice=price1+price2;
	　printf("sum=%.2f%s",sumprice,currency);
	}

2. #ifdef～#endif 形式

#ifdef~#endif 形式的条件编译的格式如下：

```
#ifdef 宏名
    程序段 1
#else
    程序段 2
#endif
```

作用：如果宏名已被#define 行定义，则编译程序段 1，否则编译程序段 2。

【说明】① #else 可以没有，但#endif 必须存在，它是#if 命令的结尾符。

②“#ifdef 宏名”的含义是判断是否定义了宏。

③ 每个命令独占一行。

【例 7.20】程序示例。

源　程　序	预处理后的新程序

#define INTEGER
#ifdef INTEGER
int add(int x, int y)//有效
{
return x+y;
}
#else
float add(float x, float y)
{
return x+y;
}
#endif
#include<stdio.h>
void main()
{
#ifdef INTEGER
int a, b, c;
scanf("%d%d", &a, &b);
printf("a+b=%d\n", add(a, b));
#else
float a, b, c;
scanf("%f%f", &a, &b);
printf("a+b=%f\n", add(a, b));
#endif
}
	int add(int x, int y)
	{
	return x+y;
	}
	#include<stdio.h>
	void main()
	{
	int a, b, c;
	scanf("%d%d", &a, &b);
	printf("a+b=%d\n", add(a, b));
	}

3. #ifndef～#endif 形式

#ifndef~#endif 形式的条件编译的格式如下：

```
#ifndef 宏名
    程序段 1
#else
    程序段 2
#endif
```

与第 2 种形式的区别仅在于：如果宏名没有被定义，则编译程序段 1，否则编译程序段 2。

4. 条件编译与分支语句的区别

① 条件编译在预编译阶段进行处理，而条件语句则在程序运行时处理。

② 条件编译中的条件不可以包含变量名，只能是常量表达式(通常包含宏名)，可以不加括号；而条件语句中的条件是条件表达式，可以包含变量或函数等，并且必须加括号。

例如：

```
#define N 10
int NUM=10;
```

```
#if  NUM==10   //错误，因为 NUM 是变量
      ……
#endif
```

③ 使用条件编译，对满足编译条件的程序语句进行编译生成目标代码，对不满足编译条件的程序语句将不进行编译；而分支语句则不管某语句是否满足条件，都要对其编译生成代码(包括分支语句本身)，所以如果用条件语句来代替条件编译命令，程序目标代码将变长。

④ 条件编译命令可以放在所有函数的外部，也可以放在某函数的内部，但分支语句只能出现在某函数内部。

习　题　7

1. 以下程序运行的结果是_____。

```
#define  ADD(x)  x+x
void  main()
{
    int  m=1, n=2, k=3;
    int  sum=ADD(m+n)*k;
    printf("sum=%d", sum);
}
```

A. sum=9　　B. sum=10　　C. sum=12　　D. sum=18

2. 以下程序运行的结果是_____。

```
#define  X  5
#define  Y  X+1
#define  Z  Y*X/2
void  main()
{
    int  a=Y;
    printf("%d, %d", Z, --a);
}
```

A. 7, 6　　B. 12, 6　　C. 12, 5　　D. 7, 5

3. 程序中头文件 type1.h 的内容是：

```
#define  N  5
#define  M1  N*3
```

程序如下：

```
#include"type1.h"
#define  M2  N*2
void  main()
{   int  i;
```

```
    i=M1+M2;
    printf("%d\n", i);
}
```

编译后运行程序，输出结果是_____。

A. 10　　B. 20　　C. 25　　D. 30

4. 下列程序中定义学生结构体变量，存储学生的学号、姓名和 3 门课的成绩。fun()函数的功能是：对形参结构体变量 a 中的数据进行修改，把修改后的数据作为函数值返回主函数进行输出。例如，若传给形参 a 的数据中，学号、姓名和三门课的成绩依次是：10001，"ZhangSan"，95，80，88，修改后的数据应为：10002，"LiSi"，96，81，89。请在下划线处填入正确的内容，使程序得出正确结果。

```
#include<stdio.h>
#include<string.h>
struct student{
    long sno;
    char name[10];
    float score[3];
};
/**********found**********/
___1___ fun(struct student a)
{   int i;
    a.sno = 10002;
    /**********found**********/
    strcpy(___2___, "LiSi");
    /**********found**********/
    for(i=0; i<3; i++) ___3___+= 1;
    return a;
}
void main()
{
    struct student s={10001, "ZhangSan", 95, 80, 88}, t;
    int i;
    printf("\n\nThe original data :\n");
    printf("\nNo: %ld Name: %s\nScores:  ", s.sno, s.name);
    for(i=0; i<3; i++)    printf("%6.2f ", s.score[i]);
    printf("\n");
    t = fun(s);
    printf("\nThe data after modified :\n");
    printf("\nNo: %ld Name: %s\nScores:  ", t.sno, t.name);
    for(i=0; i<3; i++)          printf("%6.2f ", t.score[i]);
    printf("\n");
}
```

5. 学生记录由学号和成绩组成，N 名学生的数据放入主函数中的结构体数组 s 中，请编写 fun()函数，其功能是：按分数降序(高分在前，低分在后)排列学生的记录。

6. 学生记录由学号、8 门课程成绩和平均分组成，学号和 8 门课程成绩在主函数中给出，请编写 fun()函数，其功能是：求出该学生的平均分，并放入记录的 ave 成员中。例如，学生的成绩是 85.5，76，69.5，85，91，72，64.5，87.5，则他的平均分应为 78.875。

第8章 文　　件

前面我们介绍的简单变量、数组、结构体等，都是基于内存的数据结构，程序中用到的数据，通常通过键盘输入和用显示器输出，当数据量不大时，这种方法是可行的，但是当数据量较大时，若每次运行程序时都通过键盘输入，将花费很长的时间，如果把需要处理的数据预先存储在一个磁盘文件中，当需要处理时，程序就可以从磁盘文件读取数据。同样，程序运行的结果也可以存储在磁盘文件中长期保存。

知识结构

1. 文件的基本概念
2. 文件指针
3. 文件的打开、读写与关闭
   ① 文件的打开
   ② 文件的关闭
   ③ 文件的读写
   ④ 文件读写函数的选用原则
4. 文件定位

8.1 文件的基本概念

所谓“文件”是指一组相关数据的有序集合。这个数据集有一个名称，叫做文件名。实际上，在前面的各章中我们已经多次使用过文件，例如源程序文件、目标文件、可执行文件、头文件等。文件通常驻留在外存(如磁盘等)上，在使用时才调入内存。从不同的角度可对文件作不同的分类。从用户的角度看，文件可分为普通文件和设备文件两种。

普通文件指驻留在磁盘或其他外部介质上的一个有序数据集，可以是源文件、目标文件、可执行程序，也可以是一组待输入处理的原始数据，或者是一组输出的结果。源文件、目标文件、可执行程序称为程序文件，输入输出的数据称为数据文件。

设备文件指与主机相连的各种外部设备，如显示器、打印机、键盘等。操作系统把外部设备也看作是一个文件来进行管理，把对它们的输入、输出等同于对文件的读和写。通常把显示器定义为标准输出文件，一般情况下在屏幕上显示有关信息就是向标准输出文件输出。如前面经常使用的 printf()，putchar()函数就是这类输出。键盘通常被指定为标准的输入文件，从键盘上输入就意味着从标准输入文件上输入数据。scanf()，getchar()函数就属于这类输入。

从文件编码的方式来看，可分为 ASCII 码文件和二进制码文件两种。

ASCII 码文件也称为文本文件，这种文件在磁盘中存放时每个字符对应一个字节，用于存放相应的 ASCII 码值。例如数 5678 的存储形式如图 8.1 所示，共占用 4 个字节。ASCII 码文件可在屏幕上按字符显示，例如源程序文件就是 ASCII 码文件，用编辑软件可以显示文件的内容。由于是按字符显示，因此能读懂文件内容。

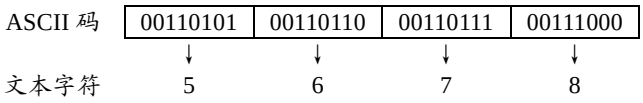


图 8.1　文本文件的存储

二进制码文件按二进制编码存放文件内容。例如数 5678 存储形式为“0001 0110 0010 1110”。 虽然也可以在屏幕上显示二进制文件的内容，但显示为乱码，无法读懂。C 语言系统在处理这些文件时，并不区分类型，都看成是字符流，按字节进行处理。输入输出字符流的开始和结束只由程序控制而不受物理符号(如回车符)的控制。 因此也把这种文件称作“流式文件”。

本章讨论流式文件的打开、关闭、读、写、定位等各种操作。

8.2　文 件 指 针

在 C 语言中用一个指针变量指向一个文件，这个指针称为文件指针。通过文件指针就可对它所指的文件进行各种操作。

1．文件结构体类型 FILE

C 语言程序在操作文件的过程中，必须保存有关文件的一些信息，比如文件名、文件状态、当前读写位置等。C 语言将这些信息保存在一个文件结构体中，在 C 语言中将这个结构体类型定义如下的 FILE 结构体。

```
typedef struct{
    short level;                /* 缓冲区满或空的程度 */
    unsigned flags;             /* 文件状态标志 */
    char fd;                    /* 文件描述符 */
    short bsize;                /* 缓冲区大小 */
    unsigned char *buffer;      /* 文件缓冲区的首地址 */
    unsigned char *curp;        /* 文件缓冲区的当前读写指针 */
    unsigned char hold;         /* 其他信息 */
    unsigned istemp;
    short token;
}FILE;
```

2．文件类型指针

在使用缓冲文件系统时，每一个文件被打开或创建之后，必须用文件类型指针作为该文件的文件标识。C 语言的编译系统中有文件结构类型 FILE 的定义，在程序中可以直接使用。

文件类型指针定义的一般格式为：

```
FILE *p;
```

这里 p 指针代表一个文件，对文件进行任何操作之前必须先定义指向文件的指针。

8.3 文件的打开、读写与关闭

在对文件进行读写操作之前要先打开文件，使用完毕要关闭文件。所谓打开文件，实际上是建立文件的各种有关信息，并使文件指针指向该文件，以便进行其他操作。关闭文件则断开指针与文件之间的联系，禁止再对该文件进行操作。

在 C 语言中，文件操作都由库函数来完成，本节将介绍主要的文件操作函数。

8.3.1 文件的打开

fopen()函数用来打开一个文件，调用 fopen()函数的一般形式为：

文件指针名=fopen(文件名, 文件打开方式);

【说明】①“文件指针名”必须是 FILE 类型的指针变量。

②“文件名”是所打开文件的文件名，可以采用相对路径，也可采用绝对路径。

③“文件打开方式”指文件的类型和操作要求。

例如：

```
FILE *fp;
fp=fopen("filea.txt", "r");
```

这段程序的意义是打开当前目录下的 filea.txt 文件，只允许进行“读”操作，并使 fp 指向该文件。

又如：

```
FILE *fp
fp=fopen("c:\\filea", "rb")
```

这段程序的意义是打开 C 驱动器磁盘的根目录下的文件 filea，这是一个二进制文件，只允许按二进制方式进行读操作。

文件打开方式用来确定对所打开的文件将进行什么操作。表 8.1 列出了 C 语言程序所有的文件打开方式。

表 8.1 文件打开方式

文本文件(ASCII 码文件)		二进制码文件	
使用方式	含 义	使用方式	含 义
"r"	打开文本文件进行只读	"rb"	打开二进制码文件进行只读
"w"	建立新文本文件进行只写	"wb"	建立新二进制码文件进行只写
"a"	打开文本文件进行追加	"ab"	打开二进制码文件进行追加
"r+"	打开文本文件进行读/写	"rb+"	打开二进制码文件进行读/写
"w+"	建立新文本文件进行读/写	"wb+"	建立新二进制码文件进行读/写
"a+"	打开文本文件进行读/写/追加	"ab+"	打开二进制码文件进行读/写/追加

具体说明如下：

① 凡用"r"方式打开一个文件时，该文件必须已经存在，否则函数返回值为 NULL。

② 用"w"方式打开的文件只能向该文件写入。若打开的文件不存在，则以指定的文件名

建立该文件，若打开的文件已经存在，则将该文件删去，重建一个新文件。

③ 若要向一个已存在的文件追加新信息，只能用"a"方式打开文件，但此时该文件必须存在，否则函数返回值为 NULL。

④ 打开一个文件时如果出错，fopen()函数将返回一个空指针值 NULL。在程序中可以用这一信息来判断是否完成了打开文件的工作，并作相应的处理。常用以下程序段打开文件：

```
if((fp=fopen("c:\\filea", "rb")==NULL)
{
    printf("\n Error on open c:\\filea file! \n");
    exit(0);
}
```

这段程序的意义是，如果返回的指针为空，表示不能打开 C 盘根目录下的 filea 文件，给出提示信息“error on open c:\\filea file!”，并结束程序运行。

8.3.2　文件的关闭

fclose()函数用来关闭文件。当完成文件操作以后，应及时关闭它，以防止不正常的操作。调用 fclose()函数的一般形式为：

```
fclose(文件指针);
```

该函数将返回一个整数，若返回为 0 表示正常关闭了文件，否则表示无法正常关闭文件，所以关闭文件操作也应该使用条件语句进行判断，常用以下程序段关闭文件：

```
if(fclose(fp))
{
    printf("\n Can not close the file! \n");
    exit(0);
}
```

执行关闭文件操作将把缓冲区中的数据强制写入磁盘，使文件指针与具体文件脱钩。这时磁盘文件和文件指针仍然存在，只是指针不再指向原来的文件。

8.3.3　文件的读写

读和写文件是最常用的文件操作。C 语言提供了多种读写文件的函数：

① 字符读写函数：fgetc()和 fputc()。

② 字符串读写函数：fgets()和 fputs()。

③ 数据块读写函数：freed()和 fwrite()。

④ 格式化读写函数：fscanf()和 fprinf()。

在程序中使用以上函数都要求包含头文件 stdio.h。

1. 字符读写函数 fgetc()和 fputc()

字符读写函数 fgetc()和 fputc()是以字符(字节)为单位的读写函数，每次从文件读出或向文件写入一个字符。

(1) 读字符函数 fgetc()

fgetc()函数的功能：从指定的文件中读一个字符，如果读取正常，返回读到的字节值并将当前读指针向后移一个字节(即指向下一个读出字符位置)；如果读到文件尾或出错，则返回 EOF(其值在头文件 stdio.h 中被定义为-1)。

调用函数的形式为：

字符变量=fgetc(文件指针);

例如：

ch=fgetc(fp);

其意义是从打开的 fp 文件中读取一个字符并送入 ch 中。

【说明】① 在调用 fgetc()函数时，读取的文件必须以读或读写方式打开。

② 读取字符的结果也可以不向字符变量赋值。例如:

fgetc(fp);

但是这样读出的字符不能保存。

【例 8.1】读入文件 a.txt，在屏幕上输出其内容。

参考代码：

```
#include<stdio.h>
#include<stdlib.h>
main()
{
    FILE  *fp;
    char  ch;
    if((fp=fopen("d:\\example\\a.txt", "r"))==NULL)
    {
        printf("\nCannot  open  file, strike  any  key  exit!");
        exit(0);
    }
    ch=fgetc(fp);
    while(ch!=EOF)
    {
        putchar(ch);
        ch=fgetc(fp);
    }
    fclose(fp);
}
```

【说明】例 8.1 程序的功能是从文件中逐个读取字符并在屏幕上显示。程序定义了文件指针 fp，以读文本文件方式打开文件 “d:\\example\\a.txt”，并使 fp 指向该文件。如打开文件出错，给出提示并退出程序。语句 “ch=fgetc(fp);”先读出一个字符，然后进入循环，只要读出的字符不是文件结束标志，就再读入下一字符。每读一次，文件内部的位置指针向后移一个字节，本程序将在屏幕上显示整个文件内容。

(2) 写字符函数 fputc()

fputc()函数的功能：将字符数据输出到文件指针所指向的文件中去，同时将读写位置指

针向后移动一个字节(即指向下一个写入位置)。如果输出成功，则函数返回值就是输出的字符数据；否则，返回 EOF。

调用 fputc()函数的一般形式为：

fputc(字符数据, 文件指量或变量);

例如：

fputc('a', fp);

其意义是将字符'a'输出到文件指针 fp 所指向的文件中去。

【例 8.2】利用字符读写函数实现文件复制。

参考代码：

```
#include<stdio.h>
#include<stdlib.h>
int main(int argc, char *argv[])
{
    FILE *input, *output;          //input：源文件指针，output：目标文件指针
    if(argc!=3)                    //参数个数不对
    {
        printf("The number of arguments not correct\n");
        printf("\n Usage：可执行文件名 source-file dest-file");
        exit(0);
    }
    if((input=fopen(argv[1], "r"))==NULL)          //如果打开源文件失败
    {
        printf("Can not open source file\n");
        exit(0);
    }
    if((output=fopen(argv[2], "w"))==NULL)         //如果创建目标文件失败
    {
        printf("Can not create destination file\n");
        exit(0);
    }
  //复制源文件到目标文件中
    for( ;(!feof(input));)
        fputc(fgetc(input), output);
    fclose(input);                                 //关闭源文件
    fclose(output);                                //关闭目标文件
    return 0;
}
```

【说明】① 例 8.2 程序的功能是将源文件的内容复制到目标文件中，源文件名和目标文件名通过命令行参数给定，加上执行文件名，命令行参数的个数应该为 3，如果不为 3 则表

示格式错误，这时给出提示信息，退出程序。

② 如果命令行格式正确，则打开源文件(用于读)和创建目标文件(用于写)，如果打开或创建失败，则给出相应提示信息，退出程序。

③ 程序中对源文件判断是否读到了文件尾，如果没有，则从源文件中读取一字节的数据写入到目标文件中，否则，文件复制结束。关闭源文件和目标文件。

在例 8.2 中用到了判断文件是否结束的库函数 feof()，其函数的原型为：

```
int feof(文件指针);
```

函数功能：在执行读文件操作时，如果遇到文件尾，则返回 1(逻辑真值)；否则，返回 0(逻辑假值)。feof()函数同时适用于 ASCII 码和二进制文件。例如：

```
!feof(input)
```

表示源文件(用于输入)未结束，可以循环读文件。

上面在介绍 fgetc()函数和 fputc()函数时，多次提到文件读写位置指针，它有什么作用？与文件指针又有什么不同呢？

在文件内部有一个位置指针，用来指向文件当前读写字节。在刚打开文件时，该指针总是指向文件的第一个字节。使用 fgetc()函数后，该位置指针将向后移动一个字节。因此可连续多次使用 fgetc()函数，读取多个字符。但应注意文件指针和文件内部的位置指针不一样。文件指针指向整个文件，须在程序中定义说明，只要不重新赋值，文件指针的值是不变的。文件内部的位置指针用来指示文件内部当前的读写位置，每读写一次，该指针均向后移动，它不需在程序中定义说明，而由系统自动设置。

2．字符串读写函数 fgets()和 fputs()

字符串读写函数 fgets()和 fputs()是以字符串的形式对文件进行读写的函数。每次可从文件读出(指定长度的字节)或向文件写入一个字符串。

(1) 读字符串函数 fgets()

函数形式为：

```
char *fgets(char *str, int num, FILE *stream);
```

功能：fgets()函数从 stream 所指向的文件中读取至多 num−1 个字符，并在其末尾加上串结束标志'\0'，把它们放入 str 指向的字符数组中。读取字符直到遇见回车符或 EOF(文件结束符)为止，或读入了所限定的字符数。

如果操作成功，返回读取的字符串的指针；如果读到文件末尾或出错，则返回 NULL。

(2) 写字符串函数 fputs()

函数形式为：

```
int fputs(char *str, FILE *stream);
```

功能：fputs()函数将 str 指向的字符串写入 stream 所指向的文件中。如果操作成功，函数返回 0 值；如果失败，则返回非零值。

【注意】使用 fputs()函数时，不会将字符串结尾符'\0'写入文件，也不会自动向文件写入换行符，如果需要写入一行文本，字符串中必须包含'\n'符。

【例 8.3】从一个文本文件 test1.txt 中读出字符串，再写入另一个文件 test2.txt。

参考代码：

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
```

```
void main()
{
    FILE *fp1, *fp2;
    char str[128];
    if((fp1=fopen("test1.txt", "r"))==NULL)      // 以只读方式打开文件 1
    {
        printf("Cannot open file\n");
        exit(0);
    }
    if((fp2=fopen("test2.txt", "w"))==NULL)      // 以只写方式打开文件 2
    {
        printf("Cannot open file\n");
         exit(0);
    }
    fgets(str, 128, fp1);                        // 把字符串写入文件 2
    fputs(str, fp2 );
    printf("%s", str);                           // 在屏幕上显示字符串
    fclose(fp1);
    fclose(fp2);
}
```

例 8.3 中的程序共操作了两个文件，需定义两个文件变量指针，在操作文件以前，应将两个文件以需要的工作方式同时打开(不分先后)，读写完成后，再关闭文件。本程序中在写入文件的同时，将写入的内容显示在屏幕上。

【例 8.4】从键盘输入一个字符串，并写入文本文件 test.txt。

参考代码：

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
void main()
{
    FILE *fp;
    char str[128];
    if((fp=fopen("test.txt", "w"))==NULL)        // 以只写方式打开文本文件
    {
        printf("Cannot open file!");
        exit(0);
    }
    while(strlen(gets(str))!=0)
    {  /*若串长度为零，则结束*/
        fputs(str, fp);                          // 写入串
```

```
            fputs("\n", fp);              // 写入回车符
        }
        fclose(fp);                       // 关闭文件
    }
```

运行例 8.4 中的程序，从键盘输入长度不超过 127 个字符的字符串，写入文件，如串长为 0(即空串)，程序结束。

这里所说的输入空串，实际为一单独的回车符，其原因是用 gets()函数输入字符串以回车作结束标志。

3. 数据块读写函数 fread()和 fwrite()

成块读写文件函数的形式为：

```
int fread(void *buf, int size, int count, FILE *stream);
int fwrite(void *buf, int size, int count, FILE *stream);
```

功能：

fread()函数表示从 stream 所指向的文件中读取 count 个数据项，每个数据项为 size 个字节，并把它们放到 buf(缓冲区)指向的字符数组中。

fwrite()函数表示从 buf(缓冲区)指向的字符数组中，把 count 个数据项写到 stream 所指向的文件中，每个数据项为 size 个字节，函数操作成功时返回所写数据项。

fread()函数返回实际已读取的数据项，如果调用函数时读取的数据项超过文件存放的数据项数，或已到文件尾则出错，实际在操作时应注意检测。

只能以二进制码文件格式创建成块读写的文件。

【例 8.5】向磁盘文件写入格式化数据，再从该文件中读出数据显示到屏幕上。

参考代码：

```
#include<stdio.h>
#include<stdlib.h>
void main()
{
    FILE *fp1;
    int i;
    struct stu                    //定义结构体变量
    {
        char name[15];
        char num[6];
        float score[2];
    }student;
    if((fp1=fopen("test.txt", "wb"))==NULL)          // 以二进制只写方式打开文件
    {
        printf("Cannot open file");
        exit(0);
    }
    printf("Input data:\n");
    for(i=0; i<2; i++)
```

```
    {
        scanf("%s %s %f %f", student.name, student.num, &student.score[0],
            &student.score[1]);                    // 输入一记录
        fwrite(&student, sizeof(student), 1, fp1);    // 成块写入文件
    }
    fclose(fp1);                                   // 关闭文件
    if((fp1=fopen("test.txt", "rb"))==NULL)        // 重新以二进制只读打开文件
    {
        printf("Cannot open file");
        exit(0);
    }
    printf("Output from file:\n");
    for(i=0; i<2; i++)
    {
        fread(&student, sizeof(student), 1, fp1);      //从文件成块读
        printf("%s %s %7.2f %7.2f\n", student.name, student.num,
            student.score[0], student.score[1]);    // 显示在屏幕上
    }
    fclose(fp1);                                   // 关闭文件
}
```

运行实例：

```
Input data:
xiaowan j001 87.5 98.4
xiaoli j002 99.5 89.6
output from file:
xiaowan j001 87.50 98.40
xiaoli j002 99.50 89.60
```

4．格式化的读写函数 fscanf()和 fprintf()

利用 scanf()和 printf()函数，可以从键盘格式化输入及在显示器上进行格式化输出。对文件格式化读写就是在上述函数的前面加一个字母 f 成为 fscanf()和 fprintf()。函数形式如下：

```
int fscanf(FILE *stream, char *format, arg_list);
int fprintf(FILE *stream, char *format, arg_list);
```

其中，stream 为文件指针，其余两个参数与 scanf()和 printf()用法完全相同。

功能：

fscanf()函数从 stream 所指向的文件中读取数据。如果操作成功，函数返回值是读取的数据项的个数；如果操作出错或遇到文件尾，则返回 EOF。

fprintf()表示将表达式输出到 stream 所指向的文件中。如果操作成功，函数返回值是写入到文件中的字节个数；如果操作出错，则返回 EOF。

例如：

```
fprintf(fp, "%d,%6.2f", i, t);      //将 i 和 t 按%d，%6.2f 格式输出到 fp 文件
fscanf(fp, "%d,%f", &i, &t);        //若文件中有 3,4.5，则将 3 送入 i，4.5 送入 t
```

【例 8.6】已知 f.txt 数据文件中保存了 5 个学生的计算机等级考试成绩，包括学号、姓名和分数，文件内容如下：

```
301101 张文 91
301102 陈慧 85
301103 王卫东 76
301104 郑伟 69
301105 郭温涛 55
```

请将文件的内容读出并显示到屏幕。

参考代码：

```
#include<stdio.h>
#include<stdlib.h>
int main(void)
{
    FILE *fp;                                   //定义文件指针
    long num;
    char stname[20];
    int score;
    if((fp=fopen("f.txt", "r"))==NULL)          //打开文件
    {
        printf("File open error\n");
        exit(0);
    }
    while(!feof(fp))
    {
        fscanf(fp, "%ld%s%d", &num, stname, &score);   //fscanf()函数读入数据
        printf("%ld  %s  %d\n", num, stname, score);   //输出到屏幕
    };
    if(fclose(fp))                                     /关闭文件
    {
        printf("Can not close the file!\n");
        exit(0);
    }
}
```

运行实例：

```
301101 张文 91
301102 陈慧 85
301103 王卫东 76
301104 郑伟 69
301105 郭温涛 55
```

例 8.6 程序中调用 fscanf()函数将文件中的数据读入到变量 num、stname 和 score，并通

过 printf()函数把它们输出到屏幕。

【总结】在 C 语言中，有两个最基本的文件操作：从文件中读取信息(读操作)和把信息存放到磁盘文件中(写操作)。为了实现读写操作，首先要定义文件指针，并打开文件，请求系统分配文件缓冲区单元，然后进行文件读写，文件操作完成后应及时关闭文件。在文件操作中，先定义指向文件的指针，然后通过调用专门的函数来实现文件的所有操作。

8.3.4　文件读写函数的选用原则

从功能角度来说，fread()和 fwrite()函数可以完成文件的任何数据读/写操作。但是为方便起见，应依下列原则选用文件读写函数：

① 读/写 1 个字符(或字节)数据时，选用 fgetc()和 fputc()函数。

② 读/写 1 个字符串时，选用 fgets()和 fputs()函数。

③ 读/写 1 个(或多个)不含格式的数据时，选用 fread()和 fwrite()函数。

④ 读/写 1 个(或多个)含格式的数据时，选用 fscanf()和 fprintf()函数。

对使用文件类型的要求：

① fgetc()和 fputc()函数主要对文本文件进行读写，但也可对二进制文件进行读写。

② fgets()和 fputs()函数主要对文本文件进行读写，对二进制文件操作无意义。

③ fread()和 fwrite()函数主要对二进制文件进行读写，但也可对文本文件进行读写。

④ fscanf()和 fprintf()函数主要对文本文件进行读写，对二进制文件操作无意义。

8.4　文 件 定 位

前面介绍的文件的字符/字符串读写，均是对文件的顺序读写，即总是从文件的开头开始进行读写。这显然不能满足我们的要求，在实际问题中经常要求只读写文件中的某一指定的部分，因此 C 语言提供了移动文件的内部指针到读写位置，再进行读写，这种读写是随机读写。实现随机读写的关键是按要求移动位置指针，这称为文件的定位。实现文件定位移动文件内部位置指针的函数主要有 rewind()、fseek()等函数。如果想知道文件读写指针的位置，可以调用 ftell()函数。

文件位置指针的最小值是 0，最大值是文件的长度。文件被打开时，文件的位置指针位于文件首部，随着数据的读写，文件的位置指针会向后移动。

下面介绍三个关于文件位置指针的函数：

```
long  ftell(FILE  *stream);
int  rewind(FILE  *stream);
fseek(FILE  *stream, long  offset, int  origin);
```

1. ftell()函数

ftell()函数功能：得到 stream 所指向文件的当前读写位置。该值是一个长整型数，用来得到文件指针离文件开头的偏移量的字节数。当返回值是-1L 时表示出错。

【例 8.7】在 file.txt 文件中，将字符串“Hello,world.”存入文件中，再检查文件指针的位置。

参考代码：

```
#include<stdio.h>
#include<stdlib.h>
int  main(void)
{
    FILE  *fp;
    long  pos;
    if((fp=fopen("file.txt", "w"))==NULL)
    {
        printf("Cannot  open  file.\n");
        exit(0);
    }
    fprintf(fp, "Hello,world.");          //向文件中写入长度为 12 的字符串
    pos=ftell(fp);
    printf("position=%ld\n", pos);
    fclose(fp);
    return  0;
}
```

运行实例：

```
position=12
```

2. **Rewind()函数**

rewind()函数功能：将文件指针移到文件的开头，如果移动成功，返回 0；否则，返回一个非 0 值。

【例 8.8】在屏幕上显示 a.txt 文件的内容，并将 a.txt 文件复制到 b.txt 文件中。

```
#include<stdio.h>
#include<stdlib.h>
int  main(void)
{
    FILE  *fp1, *fp2;
    if((fp1=fopen("a.txt", "r"))==NULL)        // 以读方式打开文本文件
    {
        printf("Cannot  open  file\n");
        exit(0);
    }
    if((fp2=fopen("b.txt", "w"))==NULL)        // 以写方式打开文本文件
    {
        printf("Cannot  open  file\n");
        exit(0);
    }
    while(!feof(fp1))                          // 在屏幕上显示文件 a.txt 的内容
        putchar(fgetc(fp1));
```

```
    rewind(fp1);                          // 使 a.txt 的位置指针重返回文件头
    while(!feof(fp1))
        fputc(fgetc(fp1), fp2);           // 把 a.txt 文件的内容复制到 b.txt 中
    fclose(fp1);
    fclose(fp2);
}
```

3. **fseek()函数**

fseek()函数功能：把文件指针以 origin 为起点移动 offset 个字节，其中 origin 指出的位置可有以下几种，如表 8.2 所示。

表 8.2　fseek 函数起点位置

Origin	数值	表示的具体位置
SEEK_SET	0	文件开头
SEEK_CUR	1	文件指针当前位置
SEEK_END	2	文件尾

例如：

```
fseek(fp, 10L, 0);
```

表示把文件指针从文件开头移到第 10 个字节处，由于 offset 参数要求是长整型数，故其数后带 L。又例如：

```
fseek(fp, -15L, 2);
```

表示把文件指针从文件尾向文件头方向移动 15 个字节。

习　题　8

1. 从键盘输入一个字符串，将其中小写字母全部转换成大写字母，然后输出到 test.txt 磁盘文件中保存。输入的字符串以“!”结束。

2. 有 A.txt 和 B.txt 两个磁盘文件，各存放一行字母，要求把这两个文件的信息合并(按字母顺序排列)，输出到一个新的 C.txt 文件中。

3. 有 5 个学生，每个学生有 3 门课程成绩，从键盘输入学生数据(包括学生号、姓名、三门课程的成绩)，计算出每个学生的平均成绩，将原有数据和计算出的平均分数存放在 stu.txt 磁盘文件中。

4. 函数调用语句“fseek(fp, -20L, 2);”的含义是_____。

A. 把文件位置指针移到距离文件头 20 个字节处

B. 把文件位置指针从当前位置向后移动 20 个字节

C. 把文件位置指针从文件末尾处向文件头方向移动 20 个字节

D. 把文件位置指针移到离当前位置 20 个字节处

5. 有以下程序：

```
#include<stdio.h>
main()
{   FILE *fp1;
    fp1=fopen("f1.txt", "w");
    fprintf(fp1, "abc");
    fclose(fp1);
}
```

若f1.txt文本文件中原有内容为good，则运行程序后，f1.txt文件中的内容为____。

A. goodabc　　B. abcd　　C. abc　　D. abcgood

6. 有以下程序：

```
#include<stdio.h>
main()
{   FILE *fp;
    int k, n, a[6]={1, 2, 3, 4, 5, 6};
    fp=fopen("d2.dat", "w");
    fprintf(fp, "%d%d%d\n", a[0], a[1], a[2]);
    fprintf(fp,"%d%d%d\n", a[3], a[4], a[5]);
    fclose(fp);
    fp=fopen("d2.dat", "r");
    fscanf(fp, "%d%d", &k, &n);
    printf("%d %d\n", k, n);
    close(fp);
}
```

运行程序后的输出结果是_____。

A. 1 2　　B. 1 4　　C. 123 4　　D. 123 456

7. 有以下程序：

```
#include<stdio.h>
main()
{   FILE *fp;    int i, k, n;
    fp=fopen("data.dat", "w+");
    for(i=1; i<6; i++)
        { fprintf(fp,"%d    ", i);
          if(i%3==0)   fprintf(fp, "\n");}
    fclose(fp);
    fp=fopen("data.dat", "r");
    fscanf(fp,"%d%d", &k, &n);   printf("%d %d\n", k, n);
    close(fp);
}
```

运行程序后的输出结果是_____。

A. 0 0　　B. 123 45　　C. 1 4　　D. 1 2

8. fp 是指向某文件的指针，且已读到此文件的末尾，则 feof(fp) 函数的返回值是_____。

A. 0　　B. NULL　　C. EOF　　D. 非零值

9. 有以下程序：

```
#include<stdio.h>
main()
{   FILE *fp;    int i;
    char ch[]="abcd", t;
    fp=fopen("abc.dat", "wb+");
```

```
    for(i=0; i<4; i++)     fwrite(&ch[i], 1, 1, fp);
    fseek(fp, -2L, SEEK_END);
    fread(&t, 1, 1, fp);
    fclose(fp);
    printf("%c\n", t);
}
```

运行程序后的输出结果是_____。

A. d　　B. c　　C. b　　D. a

10. 设有定义“FILE *fw;”，请将以下打开文件的语句补充完整，以便可以向 readme.txt 文本文件的最后续写内容。

```
fw=fopen("readme.txt", _____);
```

11. 下面程序的功能是以二进制“写”方式打开 d1.dat 文件，写入 1～100 这 100 个整数后关闭文件；再以二进制“读”方式打开 d1.dat 文件，将这 100 个整数读入到另一个数组 b 中，并显示输出。请填空。

```
#include<stdio.h>
main()
{   FILE  *fp;
    int  i, a[100], b[100];
    fp= fopen("d1.dat", "wb");
    for(i=0; i<100; i++)     a[i]=i+1;
    fwrite(a, sizeof(int), 100, fp);
    fclose(fp);
    fp=fopen("d1.dat", _________);
    fread(b, sizeof(int), 100, fp);
    fclose(fp);
    for(i=0; i<100; i++)   printf ("%d\n", b[i]);}
```

12. 已有 test.txt 文本文件，其内容为

```
Hello，everyone!
```

以下程序中，test.txt 文件已正确为“读”而打开，因此文件指针 fr 指向文件，则程序的输出结果是_____。

```
#include<stdio.h>
main()
{   FILE  *fr;     char  str[40];
    ……
    fgets(str, 5, fr);
    printf("%s\n", str);
    fclose(fr);
}
```

13. 以下各项中，与函数 fseek(fp, 0L, SEEK_SET)有相同作用的是_____。

A. feof(fp)　　B. ftell(fp)　　C. fgetc(fp)　　D. rewind(fp)

附　录

附录Ⅰ　ASCII 码表

十六进制	十进制	字符	编码	十六进制	十进制	字符	十六进制	十进制	字符	十六进制	十进制	字符
0	0		nul	20	32	空格	40	64	@	60	96	'
1	1	☺	soh	21	33	!	41	65	A	61	97	a
2	2	☻	stx	22	34	"	42	66	B	62	98	b
3	3	♥	etx	23	35	#	43	67	C	63	99	c
4	4	♦	eot	24	36	$	44	68	D	64	100	d
5	5	♣	enq	25	37	%	45	69	E	65	101	e
6	6	♠	ack	26	38	&	46	70	F	66	102	f
7	7	•	bel	27	39	`	47	71	G	67	103	g
8	8	◘	bs	28	40	(	48	72	H	68	104	h
9	9	○	ht	29	41	)	49	73	I	69	105	i
0a	10	◙	nl	2a	42	*	4a	74	J	6a	106	j
0b	11	♂	vt	2b	43	+	4b	75	K	6b	107	k
0c	12	♀	ff	2c	44	,	4c	76	L	6c	108	l
0d	13	♪	er	2d	45	-	4d	77	M	6d	109	m
0e	14	♫	so	2e	46	.	4e	78	N	6e	110	n
0f	15	☼	si	2f	47	/	4f	79	O	6f	111	o
10	16	►	dle	30	48	0	50	80	P	70	112	p
11	17	◄	dc1	31	49	1	51	81	Q	71	113	q
12	18	↕	dc2	32	50	2	52	82	R	72	114	r
13	19	‼	dc3	33	51	3	53	83	S	73	115	s
14	20	¶	dc4	34	52	4	54	84	T	74	116	t
15	21	§	nak	35	53	5	55	85	U	75	117	u
16	22	▬	syn	36	54	6	56	86	V	76	118	v
17	23	↨	etb	37	55	7	57	87	W	77	119	w
18	24	↑	can	38	56	8	58	88	X	78	120	x
19	25	↓	em	39	57	9	59	89	Y	79	121	y
1a	26	→	sub	3a	58	:	5a	90	Z	7a	122	z
1b	27	←	esc	3b	59	;	5b	91	[	7b	123	{
1c	28	∟	fs	3c	60	<	5c	92	\	7c	124	|
1d	29	↔	gs	3d	61	=	5d	93	]	7d	125	}
1e	30	▲	re	3e	62	>	5e	94	^	7e	126	~
1f	31	▼	us	3f	63	?	5f	95	_	7f	127	del

ASCII 码为 0～31(十进制)的字符是非打印字符。

附录Ⅱ　C 标准库函数

标准库中的各个函数、类型以及宏分别在以下标准头文件中说明：

<assert.h>	<float.h>	<math.h>	<stdarg.h>	<stdlib.h>
<ctype.h>	<limits.h>	<setjmp.h>	<stddef.h>	<string.h>
<errno.h>	<locale.h>	<signal.h>	<stdio.h>	<time.h>

2.1　输入与输出函数<stdio.h>

在头文件<stdio.h>中定义了用于输入和输出的函数、类型和宏。最重要的类型是用于声明文件指针的 FILE。另外两个常用的类型是 size_t 和 fpos_t，size_t 是由运算符 sizeof 产生的无符号整类型；fpos_t 类型定义能够唯一说明文件中的每个位置的对象。由头部定义的最有用的宏是 EOF，其值代表文件的结尾。

2.1.1　文件操作

① fopen

FILE *fopen(const char *filename, const char *mode);

返回：成功为 FILE 指针，失败为 NULL。

打开以 filename 所指内容为名字的文件，返回与之关联的流。mode 决定打开的方式，可选值如下：

"r"	打开文本文件用于读
"w"	创建文本文件用于写，并删除已存在的内容(如果有的话)
"a"	打开或创建文本文件用于在文件末尾写
"rb"	打开二进制文件用于读
"wb"	创建二进制文件用于写，并删除已存在的内容(如果有的话)
"ab"	打开或创建二进制文件用于在文件末尾写
"r+"	打开文本文件用于更新(即读和写)
"w+"	创建文本文件用于更新，并删除已存在的内容(如果有的话)
"a+"	打开或创建文本文件用于更新和在文件末尾写
"rb+"或"r+b"	打开二进制文件用于更新(即读和写)
"wb+"或"w+b"	创建二进制文件用于更新，并删除已存在的内容(如果有的话)
"ab+"或"a+b"	打开或创建二进制文件用于更新和在文件末尾写

后六种方式允许对同一文件进行读和写，要注意的是，在写操作和读操作的交替过程中，必须调用 fflush()或文件定位函数如 fseek()、fsetpos()、rewind()等。

文件名 filename 的长度最大为 FILENAME_MAX 个字符，一次最多可打开 FOPEN_MAX 个文件(在<stdio.h>中定义)。

② freopen

FILE *freopen(const char *filename, const char *mode, FILE *stream);

返回：成功为 stream，失败为 NULL。

以 mode 指定的方式打开文件 filename，并使该文件与流 stream 相关联。freopen()先尝试关闭与 stream 关联的文件，不管成功与否，都继续打开新文件。该函数的主要用途是把系

统定义的标准流 stdin、stdout、stderr 重定向到其他文件。

③ fflush

int fflush(FILE *stream);

返回：成功为 0，失败返回 EOF。

对输出流(写打开)，fflush()用于将已写到缓冲区但尚未写出的全部数据都写到文件中；对输入流，其结果未定义。如果写过程中发生错误则返回 EOF，正常则返回 0。fflush(NULL)用于刷新所有的输出流。程序正常结束或缓冲区满时，缓冲区自动清仓。

④ fclose

int flcose(FILE *stream);

返回：成功为 0，失败返回 EOF。

刷新 stream 的全部未写出数据，丢弃任何未读的缓冲区内的输入数据并释放自动分配的缓冲区，最后关闭流。

⑤ remove

int remove(const char *filename);

返回：成功为 0，失败为非 0 值。

删除文件 filename。

⑥ rename

int rename(const char *oldfname, const char *newfname);

返回：成功为 0，失败为非 0 值。

把文件的名字从 oldfname 改为 newfname。

⑦ tmpfile

FILE *tmpfile(void);

返回：成功为流指针，失败为 NULL。

以方式"wb+"创建一个临时文件，并返回该流的指针，该文件在被关闭或程序正常结束时被自动删除。

⑧ tmpnam

char *tmpnam(char s[L_tmpnam]);

返回：成功为非空指针，失败为 NULL。

若参数 s 为 NULL(即调用 tmpnam(NULL))，函数创建一个不同于现存文件名字的字符串，并返回一个指向一内部静态数组的指针。

若 s 非空，则函数将所创建的字符串存储在数组 s 中，并将它作为函数值返回。s 中至少要有 L_tmpnam 个字符的空间。

tmpnam 函数在每次被调用时均生成不同的名字。在程序的执行过程中，最多只能确保生成 TMP_MAX 个不同的名字。注意 tmpnam 函数只是用于创建一个名字，而不是创建一个文件。

⑨ setvbuf

int setvbuf(FILE *stream, char *buf, int mode, size_t size);

返回：成功返回 0，失败返回非 0。

控制流 stream 的缓冲区，这要在读、写以及其他任何操作之前设置。

如果 buf 非空，则将 buf 指向的区域作为流的缓冲区，如果 buf 为 NULL，函数将自行分配一个缓冲区。

size 决定缓冲区的大小。

mode 指定缓冲的处理方式，有如下值：

_IOFBF，进行完全缓冲；

_IOLBF，对文本文件表示行缓冲；

_IOLNF，不设置缓冲。

⑩ setbuf

void setbuf(FILE *stream, char *buf);

如果 buf 为 NULL，则关闭流 stream 的缓冲区；否则 setbuf 函数等价于：

(void)setvbuf(stream, buf, _IOFBF, BUFSIZ)。

注意自定义缓冲区的尺寸必须为 BUFSIZ 个字节。

2.1.2 格式化输出

① fprintf

int fprintf(FILE *stream, const char *format, …);

返回：成功为实际写出的字符数，出错返回负值。

按照 format 说明的格式把变元表中变元内容进行转换，并写入 stream 指向的流。

格式化字符串由两种类型的对象组成：普通字符(它们被拷贝到输出流)与转换规格说明(它们决定变元的转换和输出格式)。每个转换规格说明均以字符%开头，以转换字符结束。如果%后面的字符不是转换字符，那么该行为是未定义的。转换字符列表如下。

字符	说　明
d，i	int；有符号十进制表示法
O	unsigned int；无符号八进制表示法(无前导 0)
x, X	unsigned int；无符号十六进制表示法(无前导 0X 和 0x)，对 0x 用 abcdef，对 0X 用 ABCDEF
U	unsigned int；无符号十进制表示法
C	int；单个字符，转换为 unsigned char 类型后输出
S	char *；输出字符串，直到'\0'或者达到精度指定的字符数
F	double；形如[-]mmm.ddd 的十进制浮点数表示法，d 的数目由精度确定。缺省精度为 6 位，精度为 0 时不输出小数点
e，E	double；形如[-]m.dddddde[+-]xx 或者[-]m.ddddddE[+-]xx 的十进制浮点数表示法，d 的数目由精度确定。缺省精度为 6 位，精度为 0 时不输出小数点
g，G	double；当指数值小于-4 或大于等于精度时，采用%e 或%E 的格式；否则使用%f 的格式。尾部的 0 与小数点不打印
P	void *；输出指针值(具体表示与实现相关)
N	int *；到目前为止以此格式调用函数输出的字符的数目将被写入到相应变元中，不进行变元转换
%	不进行变元转换，输出%

在%与转换字符之间依次可以有下表所示的标记。

标　记	说　明
-	指定被转换的变元在其字段内左对齐
+	指定在输出的数前面加上正负号
空格	如果第一个字符不是正负号，那么在其前面附加一个空格
0	对于数值转换，在输出长度小于字段宽度时，加前导 0
#	指定其他输出格式，对于 o 格式，第一个数字必须为零；对于 x/X 格式，指定在输出的非 0 值前加 0x 或 0X；对于 e/E/f/g/G 格式，指定输出总有一个小数点；对于 g/GG

标　记	说　　明
	格式，还指定输出值后面无意义的 0 将被保留
宽度[number]	一个指定最小字段宽的数。转换后的变元输出宽度至少要达到这个数值。如果变元的字符数小于此数值，那么在变元左/右边添加填充字符。填充字符通常为空格(设置了 0 标记则为 0)
.	点号用于把字段宽和精度分开
精度[number]	对于字符串，说明输出字符的最大数目；对于 e/E/f 格式，说明输出的小数位数；对于 g/G 格式，说明输出的有效位数；对于整数，说明输出的最小位数(必要时可加前导 0)
h/l/L	长度修饰符，h 表示对应的变元按 short 或 unsigned short 类型输出；l 表示对应的变元按 long 或 unsigned long 类型输出；L 表示对应的变元按 long double 类型输出

在格式串中，字段宽度和精度二者都可以用*来指定，此时该值可通过转换对应的变元来获得，这些变元必须是 int 类型。

② printf

int printf(const char *format, …);

printf(…)等价于 fprintf(stdout, …)。

③ sprintf

int sprintf(char *buf, const char *format, …);

返回：实际写到字符数组的字符数，不包括'\0'。

与 printf()基本相同，但输出写到字符数组 buf 而不是 stdout 中，并以'\0'结束。

注意，sprintf()不对 buf 进行边界检查，buf 必须足够大，以便能装下输出结果。

④ snprintf

int snprintf(char *buf, size_t num, const char *format, …);

除了最多为 num-1 个字符被存放到 buf 指向的数组之外，snprintf()和 sprintf()完全相同。数组以'\0'结束。

该函数不属于 C89(C99 增加的)，但应用广泛，所以将其包括了进来。

⑤ vprintf，vfprintf，vsprintf，vsnprintf

int vprintf(char *format, va_list arg);

int vfprintf(FILE *stream, const char *format, va_list arg);

int vsprintf(char *buf, const char *format, va_list arg);

int vsnprintf(char *buf, size_t num, const char *format, va_list arg);

这几个函数与对应的 printf()等价，但变元表由 arg 代替。参见下面有关<stdarg.h>头文件的讨论。

vsnprintf 是 C99 增加的。

2.1.3 格式化输入

① fscanf

int fscanf(FILE *stream, const char *format, …);

返回：成功为实际被转换并赋值的输入项数目，到达文件尾或变元被转换前出错为 EOF。

在格式串 format 的控制下，从流 stream 中读入字符，把转换后的值赋给后续各个变元，在此每一个变元都必须是一个指针。当格式串 format 结束时函数返回。

格式串 format 通常包含有用于指导输入转换的转换规格说明。格式串中可以包含空格或制表符，它们将被忽略；

普通字符(%除外)，与输入流中下一个非空白字符相匹配；

转换规格说明：由一个%、一个赋值屏蔽字符*(可选)、一个用于指定最大字段宽度的数(可选)、一个用于指定目标字段的字符 h/l/L(可选)、一个转换字符组成。

转换规格说明决定了输入字段的转换方式。通常把结果保存在由对应变元指向的变量中。然而，如果转换规格说明中包含赋值屏蔽字符*，例如%*s，那么就跳过对应的输入字段，不进行赋值。输入字段是一个由非空白符组成的字符串，当遇到空白符或到达最大字段宽(如果有的话)时，对输入字段的读入结束。这意味着 scanf 函数可以跨越行的界限来读入其输入，因为换行符也是空白符(空白符包括空格、横向制表符、纵向制表符、换行符、回车符和换页符)。

转换字符列表如下。

字符	输入数据；变元类型
D	十进制整数；int *
I	整数；int *。该整数可以是以 0 开头的八进制数，也可以是以 0x/0X 开头的十六进制数
O	八进制数(可以带或不带前导 0)；unsigned int *
U	无符号十进制整数；unsigned int *
X	十六进制整数(可以带或不带前导 0x/0X)；unsigned int *
C	字符；char *。按照字段宽的大小把读入的字符保存在指定的数组中，不加入字符'\0'。字段宽的缺省值为 1。在这种情况下，不跳过空白符；如果要读入下一个非空白符，使用%1s(数字 1)
S	由非空白符组成的字符串(不包含引号)；char *。该变元指针指向一个字符数组，该字符数组有足够空间来保存该字符串以及在末尾添加的'\0'
e/f/g	浮点数；float *。float 浮点数的输入格式为：一个任选的正负号，一串可能包含小数点的数字和一个任选的指数字段。指数字段由字母 e/E 以及后跟的一个可能带正负号的整数组成
P	用 printf("%p")调用输出的指针值；void *
N	将到目前为止调用所读的字符数写入变元；int *。不读入输入字符。不增加转换项目计数
[…]	用方括号括起来的字符集中的字符来匹配输入，以找到最长的非空字符串；char *。在末尾添加'\0'。格式[]…]表示字符集中包含字符]
[…]	用不在方括号里的字符集中的字符来匹配输入，以找到最长的非空字符串；char *。在末尾添加'\0'。格式[]...]表示字符集中包含字符]
%	字面值%，不进行赋值

字段类型字符：

如果变元是指向 short 类型而不是 int 类型的指针，那么在 d/i/n/o/u/x 这几个转换字符前可以加上字符 h；

如果变元是指向 long 类型的指针，那么在 d/i/n/o/u/x 这几个转换字符前可以加上字符 l；

如果变元是指向 double 类型而不是 float 类型的指针，那么在 e/f/g 这几个转换字符前可以加上字符 l；

如果变元是指向 long double 类型的指针，那么在 e/f/g 前可以加上字符 L。

② scanf

int scanf(const char *format, …);

scanf(…)等价于 fscanf(stdin, …)。

③ sscanf

int sscanf(const char *buf, const char *format, …);

与 scanf()基本相同，但 sscanf()从 buf 指向的数组中读，而不是 stdin。

2.1.4 字符输入输出函数

① fgetc

int fgetc(FILE *stream);

以 unsigned char 类型返回输入流 stream 中下一个字符(转换为 int 类型)。如果到达文件末尾或发生错误，则返回 EOF。

② fgets

char *fgets(char *str, int num, FILE *stream);

返回：成功返回 str，到达文件尾或发生错误返回 NULL。

从流 stream 中读入最多 num−1 个字符到数组 str 中。当遇到换行符时，把换行符保留在 str 中，读入不再进行。数组 str 以'\0'结尾。

③ fputc

int fputc(int ch, FILE *stream);

返回：成功为所写的字符，出错为 EOF。

把字符 ch(转换为 unsigned char 类型)输出到流 stream 中。

④ fputs

int fputs(const char *str, FILE *stream);

返回：成功返回非负值，失败返回 EOF。

把字符串 str 输出到流 stream 中，不输出终止符'\0'。

⑤ getc

int getc(FILE *stream);

getc()与 fgetc()等价。不同之处为：当 getc 函数被定义为宏时，它可能多次计算 stream 的值。

⑥ getchar

int getchar(void);

等价于 getc(stdin)。

⑦ gets

char *gets(char *str);

返回：成功为 str，到达文件尾或发生错误则为 NULL。

从 stdin 中读入下一个字符串到数组 str 中，并把读入的换行符替换为字符'\0'。

gets()可读入无限多字节，所以要保证 str 有足够的空间，防止溢出。

⑧ putc

int putc(int ch, FILE *stream);

putc()与 fputc()等价。不同之处为：当 putc 函数被定义为宏时，它可能多次计算 stream 的值。

⑨ putchar

int putchar(int ch);

等价于 putc(ch, stdout)。

⑩ puts

int puts(const char *str);

返回：成功返回非负值，出错返回 EOF。

把字符串 str 和一个换行符输出到 stdout。

⑪ ungetc

int ungetc(int ch, FILE *stream);

返回：成功时为 ch，出错为 EOF。

把字符 ch(转换为 unsigned char 类型)写回到流 stream 中，下次对该流进行读操作时，将返回该字符。对每个流只保证能写回一个字符(有些实现支持回退多个字符)，且此字符不能是 EOF。

2.1.5　直接输入输出函数

① fread

size_t fread(void *buf, size_t size, size_t count, FILE *stream);

返回：实际读入的对象数。

从流 stream 中读入最多 count 个长度为 size 个字节的对象，放到 buf 指向的数组中。

返回值可能小于指定读入数，原因可能是出错，也可能是到达文件尾。实际执行状态可用 feof()或 ferror()确定。

② fwrite

size_t fwrite(const void *buf, size_t size, size_t count, FILE *stream);

返回：实际输出的对象数。

把 buf 指向的数组中 count 个长度为 size 的对象输出到流 stream 中，并返回被输出的对象数。如果发生错误，则返回一个小于 count 的值。

2.1.6　文件定位函数

① fseek

int fseek(FILE *stream, long int offset, int origin);

返回：成功为 0，出错为非 0。

对与流 stream 相关的文件定位，随后的读写操作将从新位置开始。

对于二进制文件，此位置被定位在由 origin 开始的 offset 个字符处。origin 的值可能为 SEEK_SET(文件开始处)、SEEK_CUR(当前位置)或 SEEK_END(文件结束处)。

对于文本流，offset 心须为 0，或者是由函数 ftell()返回的值(此时 origin 的值必须是 SEEK_SET)。

② ftell

long int ftell(FILE *stream);

返回与流 stream 相关的文件的当前位置。出错时返回-1L。

③ rewind

void rewind(FILE *stream);

rewind(fp)等价于 fssek(fp, 0L, SEEK_SET)与 clearerr(fp)这两个函数顺序执行的效果，即把与流 stream 相关的文件的当前位置移到开始处，同时清除与该流相关的文件尾标志和错误标志。

④ fgetpos

int fgetpos(FILE *stream, fpos_t *position);

返回：成功返回 0，失败返回非 0。

把流 stream 的当前位置记录在*position 中，供随后的 fsetpos()调用时使用。

⑤ fsetpos

int fsetpos(FILE *stream, const fpos_t *position);

返回：成功返回 0，失败返回非 0。

把流 stream 的位置定位到*position 中记录的位置。*position 的值是之前调用 fgetpos()记录下来的。

2.1.7 错误处理函数

当发生错误或到达文件末尾时，标准库中的许多函数将设置状态指示符。这些状态指示符可被显式地设置和测试。另外，整数表达式 errno(定义在<errno.h>中的)可包含一个出错序号，该数将进一步给出最近一次出错的信息。

① clearerr

void clearerr(FILE *stream);

清除与流 stream 相关的文件结束指示符和错误指示符。

② feof

int feof(FILE *stream);

返回：到达文件尾时返回非 0 值，否则返回 0。

与流 stream 相关的文件结束指示符被设置时，函数返回一个非 0 值。

③ ferror

int ferror(FILE *stream);

返回：无错返回 0，有错返回非 0。

与流 stream 相关的文件出错指示符被设置时，函数返回一个非 0 值。

④ perror

void perror(const char *str);

perror(s)用于输出字符串 s 以及与全局变量 errno 中的整数值相对应的出错信息，具体出错信息的内容依赖于实现。该函数的功能类似于：

fprintf(stderr, "%s: %s\n", s, "出错信息");

可参见下面第 2.3 节介绍的 strerror 函数。

2.2 字符类测试函数<ctype.h>

在头文件<ctype.h>中说明了一些用于测试字符的函数。每个函数的变元均为 int 类型，变元的值必须是 EOF 或可用 unsigned char 类型表示的字符，函数的返回值为 int 类型。如果变元满足所指定的条件，那么函数返回非 0 值(表示真)；否则返回值为 0(表示假)。

在 7 位 ASCII 字符集中，可打印字符是从 0x20(' ')到 0x7E('~')之间的字符；控制字符是从 0(NUL)到 0x1F(US)之间的字符和字符 0x7F(DEL)。

① isalnum

int sialnum(int ch);

变元为字母或数字时，函数返回非 0 值，否则返回 0。

② isalpha

int isalpha(int ch);

当变元为字母表中的字母时，函数返回非 0 值，否则返回 0。

各种语言的字母表互不相同，对英语来说，字母表由大写字母 A 到 Z 和小写字母 a 到 z 组成。

③ iscntrl

int iscntrl(int ch);

当变元是控制字符时，函数返回非 0 值，否则返回 0。

④ isdigit

int isdigit(int ch);

当变元是十进制数字时，函数返回非 0 值，否则返回 0。

⑤ isgraph

int isgraph(int ch);

如果变元为除空格之外的任何可打印字符，函数返回非 0 值，否则返回 0。

⑥ islower

int islower(int ch);

如果变元是小写字母，函数返回非 0 值，否则返回 0。

⑦ isprint

int isprint(int ch);

如果变元是可打印字符(含空格)，函数返回非 0 值，否则返回 0。

⑧ ispunct

int ispunct(int ch);

如果变元是除空格、字母和数字外的可打印字符，函数返回非 0 值，否则返回 0。

⑨ isspace

#include <ctype.h> int isspace(int ch);

当变元为空白字符(包括空格、换页符、换行符、回车符、水平制表符和垂直制表符)时，函数返回非 0 值，否则返回 0。

⑩ isupper

#include <ctype.h> int isupper(int ch);

如果变元为大写字母，函数返回非 0 值，否则返回 0。

⑪ isxdigit

int isxdigit(int ch);

当变元为十六进制数字时，函数返回非 0 值，否则返回 0。

⑫ tolower

int tolower(int ch);

当 ch 为大写字母时，返回其对应的小写字母；否则返回 ch。

⑬ toupper

int toupper(int ch);

当 ch 为小写字母时，返回其对应的大写字母；否则返回 ch。

2.3　字符串函数<string.h>

在头文件<string.h>中定义了两组字符串函数。第一组函数的名字以 str 开头；第二组函数的名字以 mem 开头。只有函数 memmove 对重叠对象间的拷贝进行了定义，而其他函数都未定义。比较类函数将其变元视为 unsigned char 类型的数组。

① strcpy

char *strcpy(char *str1, const char *str2);

把字符串 str2(包括'\0')拷贝到字符串 str1 当中，并返回 str1。

② strncpy

char *strncpy(char *str1, const char *str2, size_t count);

把字符串 str2 中最多 count 个字符拷贝到字符串 str1 中，并返回 str1。如果 str2 中少于 count 个字符，那么就用'\0'来填充，直到满足 count 个字符为止。

③ strcat

char *strcat(char *str1, const char *str2);

把 str2(包括'\0')拷贝到 str1 的尾部(连接)，并返回 str1。其中原 str1 的终止符'\0'被 str2 的第一个字符覆盖。

④ strncat

char *strncat(char *str1, const char *str2, size_t count);

把 str2 中最多 count 个字符连接到 str1 的尾部，并以'\0'终止 str1，返回 str1。其中原 str1 的终止符'\0'被 str2 的第一个字符覆盖。

注意，最大拷贝字符数是 count+1。

⑤ strcmp

int strcmp(const char *str1, const char *str2);

按字典顺序比较两个字符串，返回整数值的意义如下：

小于 0，str1 小于 str2；

等于 0，str1 等于 str2；

大于 0，str1 大于 str2。

⑥ strncmp

int strncmp(const char *str1, const char *str2, size_t count);

同 strcmp，除了最多比较 count 个字符。根据比较结果返回的整数值如下：

小于 0，str1 小于 str2；

等于 0，str1 等于 str2；

大于 0，str1 大于 str2。

⑦ strchr

char *strchr(const char *str, int ch);

返回指向字符串 str 中字符 ch 第一次出现的位置的指针，如果 str 中不包含 ch，则返回 NULL。

⑧ strrchr

char *strrchr(const char *str, int ch);

返回指向字符串 str 中字符 ch 最后一次出现的位置的指针，如果 str 中不包含 ch，则返回 NULL。

⑨ strspn

size_t strspn(const char *str1, const char *str2);

返回字符串 str1 中由字符串 str2 中字符构成的第一个子串的长度。

⑩ strcspn

size_t strcspn(const char *str1, const char *str2);

返回字符串 str1 中由不在字符串 str2 中字符构成的第一个子串的长度。

⑪ strpbrk

char *strpbrk(const char *str1, const char *str2);

返回指向字符串 str2 中的任意字符第一次出现在字符串 str1 中的位置的指针；如果 str1 中没有与 str2 相同的字符，则返回 NULL。

⑫ strstr

char *strstr(const char *str1, const char *str2);

返回指向字符串 str2 第一次出现在字符串 str1 中的位置的指针；如果 str1 中不包含 str2，则返回 NULL。

⑬ strlen

size_t strlen(const char *str);

返回字符串 str 的长度，'\0'不算在内。

⑭ strerror

char *strerror(int errnum);

返回指向与错误序号 errnum 对应的错误信息字符串的指针(错误信息的具体内容依赖于实现)。

⑮ strtok

char *strtok(char *str1, const char *str2);

在 str1 中搜索由 str2 中的分界符界定的单词。

对 strtok()的一系列调用将把字符串 str1 分成许多单词，这些单词以 str2 中的字符为分界符。第一次调用时，str1 非空，它搜索 str1，找出由非 str2 中的字符组成的第一个单词，将 str1 中的下一个字符替换为'\0'，并返回指向单词的指针。随后的每次 strtok()调用(参数 str1 用 NULL 代替)，均从前一次结束的位置之后开始，返回下一个由非 str2 中的字符组成的单词。当 str1 中没有这样的单词时返回 NULL。每次调用时，字符串 str2 可以不同。

如：

```
char *p; p = strtok("The summer soldier,the sunshine patriot", " ");
printf("%s", p);
do{ p = strtok("\0", ", ");                    /* 此处 str2 是逗号和空格 */
    if(p) printf("|%s", p);}
while (p);
//显示结果是：The | summer | soldier | the | sunshine | patriot
```

⑯ memcpy

void *memcpy(void *to, const void *from, size_t count);

把 from 中的 count 个字符拷贝到 to 中，并返回 to。

⑰ memmove

void *memmove(void *to, const void *from, size_t count);

功能与 memcpy 类似，不同之处在于，当发生对象重叠时，函数仍能正确执行。

⑱ memcmp

int memcmp(const void *buf1, const void *buf2, size_t count);

比较 buf1 和 buf2 的前 count 个字符，返回值与 strcmp 的返回值相同。

⑲ memchr

void *memchr(const void *buffer, int ch, size_t count);

返回指向 ch 在 buffer 中第一次出现的位置指针，如果在 buffer 的前 count 个字符当中找不到匹配字符，则返回 NULL。

⑳ memset

void *memset(void *buf, int ch, size_t count);

把 buf 中的前 count 个字符替换为 ch，并返回 buf。

2.4 数学函数<math.h>

在头文件<math.h>中说明了数学函数和宏。

宏 EDOM 和 ERANGE(定义在头文件<errno.h>中)是两个非 0 整常量，用于引发调用各个数学函数时发生的定义域错误和值域错误；HUGE_VAL 是一个 double 类型的正数。当变元在函数的定义域之外取值时，就会出现定义域错误。在发生定义域错误时，全局变量 errno 的值被置为 EDOM，函数的返回值视具体实现而定。如果函数的结果不能用 double 类型表示，那么就会发生值域错误。当结果上溢时，函数返回 HUGE_VAL 并带有正确的符号(正负号)，errno 的值被置为 ERANGE。当结果下溢时，函数返回 0，而 errno 是否被设置为 ERANGE 视具体实现而定。

① sin

double sin(double arg);

返回 arg 的正弦值，arg 单位为弧度。

② cos

double cos(double arg);

返回 arg 的余弦值，arg 单位为弧度。

③ tan

double tan(double arg);

返回 arg 的正切值，arg 单位为弧度。

④ asin

double asin(double arg);

返回 arg 的反正弦值，值域为[−pi/2, pi/2]，其中变元范围为[−1, 1]。

⑤ acos

double acos(double arg);

返回 arg 的反余弦值，值域为[0, pi]，其中变元范围为[−1,1]。

⑥ atan

double atan(double arg);

返回 arg 的反正切值，值域为[−pi/2, pi/2]。

⑦ atan2

double atan2(double a, double b);

返回 a/b 的反正切值 $\tan^{-1}(a/b)$，值域为[−pi, pi]。

⑧ sinh

double sinh(double arg);

返回 arg 的双曲正弦值。

⑨ cosh

double cosh(double arg);

返回 arg 的双曲余弦值。

⑩ tanh

double tanh(double arg);

返回 arg 的双曲正切值。

⑪ exp

double exp(double arg);

返回以 e 为底数的 arg 的幂函数值。

⑫ log

double log(double arg);

返回 arg 的自然对数值，其中变元范围 arg > 0。

⑬ log10

double log10(double arg);

返回以 10 为底的 arg 的对数，其中变元范围 arg > 0。

⑭ pow

double pow(double x, double y);

返回 x^y，如果 x=0 且 y<=0 或者如果 x<0 且 y 不是整数，则产生定义域错误。

⑮ sqrt

double sqrt(double arg);

返回 arg 的平方根，其中变元范围 arg>=0。

⑯ ceil

double ceil(double arg);

返回不小于 arg 的最小整数。

⑰ floor

double floor(double arg);

返回不大于 arg 的最大整数。

⑱ fabs

double fabs(double arg);

返回 arg 的绝对值。

⑲ ldexp

double ldexp(double num, int exp);

返回 num * 2exp。

⑳ frexp

double frexp(double num, int *exp);

把 num 分成一个在[1/2, 1)区间的真分数和一个 2 的幂数。将真分数返回，幂数保存在*exp 中。如果 num 等于 0，那么这两部分均为 0。

㉑ modf

double modf(double num, double *i);

把 num 分成整数和小数两部分，两部分均与 num 有同样的正负号。函数返回小数部分，整数部分保存在*i 中。

㉒ fmod

double fmod(double a, double b);

返回 a/b 的浮点余数，符号与 a 相同。如果 b 为 0，那么结果由具体实现而定。

2.5 实用函数<stdlib.h>

在头文件<stdlib.h>中说明了用于数值转换、内存分配以及具有其他相似任务的函数。

① atof

double atof(const char *str);

把字符串 str 转换成 double 类型。等价于：strtod(str, (char**)NULL)。

② atoi

int atoi(const char *str);

把字符串 str 转换成 int 类型。等价于：(int)strtol(str, (char**)NULL, 10)。

③ atol

long atol(const char *str);

把字符串 str 转换成 long 类型。等价于：strtol(str, (char**)NULL, 10)。

④ strtod

double strtod(const char *start, char **end);

把字符串 start 的前缀转换成 double 类型。在转换中跳过 start 的前导空白符，然后逐个读入构成数的字符，任何非浮点数成分的字符都会终止上述过程。如果 end 不为 NULL，则把未转换部分的指针保存在*end 中。

如果结果上溢，返回带有适当符号的 HUGE_VAL，如果结果下溢，则返回 0。在这两种情况下，errno 均被置为 ERANGE。

⑤ strtol

long int strtol(const char *start, char **end, int radix);

把字符串 start 的前缀转换成 long 类型，在转换中跳过 start 的前导空白符。如果 end 不为 NULL，则把未转换部分的指针保存在*end 中。

如果 radix 的值在 2 到 36 之间，那么转换按该基数进行；如果 radix 为 0，则基数为八进制、十进制、十六进制，以 0 为前导的是八进制，以 0x 或 0X 为前导的是十六进制。无论在哪种情况下，串中的字母都是表示 10 到 radix-1 之间数字的字母。如果 radix 是 16，可以加上前导 0x 或 0X。

如果结果上溢，则依据结果的符号返回 LONG_MAX 或 LONG_MIN，置 errno 为 ERANGE。

⑥ strtoul

unsigned long int strtoul(const char *start, char **end, int radix);

与 strtol()函数类似，只是结果为 unsigned long 类型，溢出时值为 ULONG_MAX。

⑦ rand

int rand(void);

产生一个 0 到 RAND_MAX 之间的伪随机整数。RAND_MAX 值至少为 32767。

⑧ srand

void srand(unsigned int seed);

设置新的伪随机数序列的种子为 seed。种子的初值为 1。

⑨ calloc

void *calloc(size_t num, size_t size);

为 num 个大小为 size 的对象组成的数组分配足够的内存，并返回指向所分配区域的第一个字节的指针；如果内存不足以满足要求，则返回 NULL。

分配的内存区域中的所有位被初始化为 0。

⑩ malloc

void *malloc(size_t size);

为大小为 size 的对象分配足够的内存，并返回指向所分配区域的第一个字节的指针；如果内存不足以满足要求，则返回 NULL。

对分配的内存区域不进行初始化。

⑪ realloc

void *realloc(void *ptr, size_t size);

将 ptr 指向的内存区域的大小改为 size 个字节。如果新分配的内存比原内存大，那么原内存的内容保持不变，增加的空间不进行初始化。如果新分配的内存比原内存小，那么新内存保持原内存区中前 size 字节的内容。函数返回指向新分配空间的指针。如果不能满足要求，则返回 NULL，原 ptr 指向的内存区域保持不变。

如果 ptr 为 NULL，则函数行为等价于 malloc(size)。

如果 size 为 0，则函数行为等价于 free(ptr)。

⑫ free

void free(void *ptr);

释放 ptr 指向的内存空间，若 ptr 为 NULL，则什么也不做。ptr 必须指向先前用动态分配函数 malloc()、realloc()或 calloc()分配的空间。

⑬ abort

void abort(void);

使程序非正常终止。其功能类似于 raise(SIGABRT)。

⑭ exit

void exit(int status);

使程序正常终止。exit 函数以与注册相反的顺序被调用，所有打开的文件被刷新，所有打开的流被关闭。status 的值如何被返回依具体的实现而定，但用 0 表示正常终止，也可用值 EXIT_SUCCESS 和 EXIT_FAILURE 表示。

⑮ atexit

int atexit(void (*func)(void));

注册在程序正常终止时所要调用的函数 func。如果成功注册，则函数返回 0 值，否则返回非 0 值。

⑯ system

int system(const char *str);

把字符串 str 传送给执行环境。如果 str 为 NULL，那么在存在命令处理程序时，返回 0 值。如果 str 的值非 NULL，则返回值与具体的实现有关。

⑰ getenv

char *getenv(const char *name);

返回与 name 相关的环境字符串。如果该字符串不存在，则返回 NULL。其细节与具体的实现有关。

⑱ bsearch

void *bsearch(const void *key, const void *base, size_t n, size_t size,\
int (*compare)(const void *, const void *));

在 base[0]…base[n-1]之间查找与*key 匹配的项。size 指出每个元素占有的字节数。函数返回一个指向匹配项的指针，若不存在匹配，则返回 NULL。

函数指针 compare 指向的函数把关键字 key 和数组元素比较，比较函数的形式为：

int func_name(const void *arg1, const void *arg2);

arg1 是 key 指针，arg2 是数组元素指针。

返回值必须如下：

arg1 < arg2 时，返回值<0。

arg1 = arg2 时，返回值=0。

arg1 > arg2 时，返回值>0。

数组 base 必须按升序排列(与 compare 函数定义的大小次序一致)。

⑲ qsort

void qsort(void *base, size_t n, size_t size, \
int (*compare)(const void *, const void *));

对由 n 个大小为 size 的对象构成的数组 base 进行升序排序。

比较函数 compare 的形式如下：

int func_name(const void *arg1, const voie *arg2);

其返回值必须如下所示：

arg1 < arg2，返回值<0。

arg1 = arg2，返回值=0。

arg1 > arg2，返回值>0。

⑳ abs

int abs(int num);

返回 int 变元 num 的绝对值。

㉑ labs

long labs(long int num);

返回 long 类型变元 num 的绝对值。

㉒ div

div_t div(int numerator, int denominator);

返回 numerator/denominator 的商和余数，结果分别保存在结构类型 div_t 的两个 int 成员 quot 和 rem 中。

㉓ ldiv

ldiv_t div(long int numerator, long int denominator);

返回 numerator/denominator 的商和余数，结果分别保存在结构类型 ldiv_t 的两个 long 成员 quot 和 rem 中。

2.6 诊断函数<assert.h>

void assert(int exp);

assert 宏用于为程序增加诊断功能。当 assert(exp)执行时，如果 exp 为 0，则在标准出错

输出流 stderr 输出一条如下所示的信息：

Assertion failed: expression, file filename, line nnn

然后调用 abort 终止执行。其中的源文件名 filename 和行号 nnn 来自于预处理宏__FILE__和__LINE__。

如果<assert.h>被包含时定义了宏 NDEBUG，那么宏 assert 被忽略。

2.7 变长变元表函数<stdarg.h>

头文件<stdarg.h>中的说明提供了依次处理含有未知数目和类型的函数变元表的机制。

① va_start

```
void va_start(va_list ap, lastarg);
```

② va_arg

```
type va_arg(va_list ap, type);
```

③ va_end

```
void va_end(va_list ap);
```

假设函数 f 含有可变数目的变元，lastarg 是它的最后一个有名参数，然后在 f 内说明一个类型为 va_list 的变量 ap，它将依次指向每个变元：

```
va_list ap;
```

在访问任何未命名的变元前必须用 va_start 宏对 ap 进行初始化：

```
va_start(ap, lastarg);
```

此后，宏 va_arg 的每次执行将产生一个与下一个未命名的变元有相同类型和值的值，它同时还修改 ap，以使下一次使用 va_arg 时返回下一个变元：

```
va_arg(ap, type);
```

当所有的变元处理完毕之后，f 返回之前，必须调用一次宏 va_end：

```
va_end(ap);
```

例子：函数 sum_series()的第一个参数是变元项数。

```
double sum_series(int num, …)
{   double sum = 0.0, t;
    va_list ap;
    va_start(ap, num);
    for( ; num; num--)
      { t = va_arg(ap, double);   sum += t; }
    va_end(ap);
    return sum; }
```

2.8 非局部跳转函数<setjmp.h>

头文件<setjmp.h>中的说明提供了一种避免通常的函数调用和返回顺序的途径，特别的，它允许立即从一个多层嵌套的函数调用中返回。

① setjmp

```
int setjmp(jmp_buf env);
```

setjmp()宏把当前状态信息保存到 env 中，供以后 longjmp()恢复状态信息时使用。如果是直接调用 setjmp()，那么返回值为 0；如果是由于调用 longjmp()而调用 setjmp()，那么返

回值非 0。setjmp()只能在某些特定情况下调用，如在 if 语句、switch 语句、循环语句的条件测试部分以及一些简单的关系表达式中。

② longjmp

void longjmp(jmp_buf env, int val);

longjmp()用于恢复由最近一次调用 setjmp()时保存到 env 的状态信息。当它执行完时，程序就像 setjmp()刚刚执行完并返回非 0 值 val 那样继续执行。包含 setjmp()宏调用的函数一定不能已经终止。所有可访问的对象的值都与调用 longjmp()时相同；唯一的例外是，那些调用 setjmp()宏的函数中的非 volatile 自动变量如果在调用 setjmp()后有了改变，那么就变成未定义的。

2.9 信号处理函数<signal.h>

在头文件<signal.h>中提供了一些用于处理程序运行期间所引发的异常条件的功能，如处理来源于外部的中断信号或程序执行期间出现的错误等事件。

① signal

void(*signal(int sig, void(*handler)(int)))(int);

signal()用于确定以后当信号 sig 出现时的处理方法。如果 handler 的值是 SIG_DFL，那么就采用实现定义的缺省行为；如果 handler 的值是 SIG_IGN，那么就忽略该信号；否则，调用 handler 所指向的函数(参数为信号类型)。有效的信号如下表所示。

SIGABRT	异常终止，如调用 abort()
SIGFPE	算术运算出错，如除数为 0 或溢出
SIGILL	非法函数映象，如非法指令
SIGINT	交互式信号，如中断
SIGSEGV	非法访问存储器，如访问不存在的内存单元
SIGTERM	发送给本程序的终止请求信号

signal()返回信号 sig 原来的 handler；如果出错，则返回 SIG_ERR。

当随后出现信号 sig 时，就中断正在执行的操作，转而执行信号处理函数(*handler)(sig)。如果从信号处理程序中返回，则从中断的位置继续执行。

信号的初始状态由实现定义。

② raise

int raise(int sig);

向程序发送信号 sig。如果发送不成功，就返回一个非 0 值。

2.10 日期与时间函数<time.h>

在头文件<time.h>中说明了一些用于处理日期和时间的类型和函数。其中的一部分函数用于处理当地时间，因为时区等原因，当地时间与日历时间可能不相同。clock_t 和 time_t 是两个用于表示时间的算术类型数据，而 struct tm 则用于存放日历时间的各个成分。tm 的各个成员的用途及取值范围如下：

```
int tm_sec; /* 秒，0～61 */
int tm_hour; /* 时，0～23 */
int tm_mon; /* 月(从 1 月开始)，0～11 */
int tm_wday; /* 星期(从周日开始)，0～6 */
int tm_min; /* 分，0～59 */
int tm_mday; /* 日，1～31 */
int tm_year; /* 年(从 1900 年开始) */
```

int tm_yday; /* 天数(从 1 月 1 日开始)，0～365 */
int tm_isdst; /* 夏令时标记 */

其中，tm_isdst 在使用夏令时时其值为正，在不使用夏令时时其值为 0，如果该信息不能使用，其值为负。

① clock

clock_t clock(void);

返回程序自开始执行到目前为止所占用的处理机时间。如果处理机时间不可使用，那么返回-1。clock()/CLOCKS_PER_SEC 是以秒为单位表示的时间。

② time

time_t time(time_t *tp);

返回当前日历时间。如果日历时间不能使用，则返回-1。如果 tp 不为 NULL，那么同时把返回值赋给*tp。

③ difftime

double difftime(time_t time2, time_t time1);

返回 time2-time1 的值(以秒为单位)。

④ mktime

time_t mktime(struct tm *tp);

将结构*tp 中的当地时间转换为 time_t 类型的日历时间，并返回该时间。如果不能转换，则返回-1。

⑤ asctime

char *asctime(const struct tm *tp);

将结构*tp 中的时间转换成如下所示的字符串形式：

day month date hours:minutes:seconds year\n\0

如：Fri Apr 15 15:14:13 2005\n\0

返回指向该字符串的指针。字符串存储在可被其他调用重写的静态对象中。

⑥ ctime

char *ctime(const time_t *tp);

将*tp 中的日历时间转换为表示当地时间的字符串，并返回指向该字符串指针。字符串存储在可被其他调用重写的静态对象中。等价于如下调用：

asctime(localtime(tp))

⑦ gmtime

struct tm *gmtime(const time_t *tp);

将*tp 中的日历时间转换成 struct tm 结构形式的国际标准时间(UTC)，并返回指向该结构的指针。如果转换失败，则返回 NULL。结构内容存储在可被其他调用重写的静态对象中。

⑧ localtime

struct tm *localtime(const time_t *tp);

将*tp 中的日历时间转换成 struct tm 结构形式的本地时间，并返回指向该结构的指针。结构内容存储在可被其他调用重写的静态对象中。

⑨ strftime

size_t strftime(char *s, size_t smax, const char *fmt, const struct tm *tp);

根据 fmt 的格式说明，把结构*tp 中的日期与时间信息转换成指定的格式，并存储到 s 所

指向的数组中，写到 s 中的字符数不能多于 smax。函数返回实际写到 s 中的字符数(不包括'\0')；如果产生的字符数多于 smax，则返回 0。

fmt 类似于 printf()中的格式说明，它由 0 个或多个转换规格说明与普通字符组成。普通字符原封不动的拷贝到s 中,每个%c按照下表所描述的格式用与当地环境相适应的值来替换。

格式	说　明
%a	一星期中各天的缩写名
%A	一星期中各天的全名
%b	缩写月份名
%B	月份全名
%c	当地时间和日期表示
%d	用整数表示的一个月中的第几天(01～31)
%H	用整数表示的时(24 小时制，00～23)
%I	用整数表示的时(12 小时制，01～12)
%j	用整数表示的一年中各天(001～366)
%m	用整数表示的月份(01～12)
%M	用整数表示的分(00～59)
%p	与 AM/PM 对应的当地表示方法
%S	用整数表示的秒(00～61)
%U	用整数表示一年中的星期数(00～53，将星期日看作每周的第一天)
%w	用整数表示一周中的各天(0～6，星期日为 0)
%W	用整数表示一年中的星期数(00～53，将星期一看作每周的第一天)
%x	当地日期表示
%X	当地时间表示
%y	不带公元的年(00～99)
%Y	完整年份表示
%Z	时区名字(可获得时)
%%	%本身

2.11　由实现定义的限制<limits.h>和<float.h>

在头文件<limits.h>中定义了用于表示整类型大小的常量。下表所列的值是可接受的最小值，实际系统中可能有更大的值。

CHAR_BIT	8	char 类型的位数
CHAR_MAX	UCHAR_MAX 或 SCHAR_MAX	char 类型的最大值
CHAR_MIN	0 或 SCHAR_MIN	char 类型的最小值
INT_MAX	32767	int 类型的最大值
INT_MIN	−32767	int 类型的最小值
LONG_MAX	2147483647	long 类型的最大值
LONG_MIN	−2147483647	long 类型的最小值
SCHAR_MAX	+127	signed char 类型的最大值
SCHAR_MIN	−127	signed char 类型的最小值
SHRT_MAX	+32767	short 类型的最大值
SHRT_MIN	−32767	short 类型的最小值
UCHAR_MAX	255	unsigned char 类型的最大值
UINT_MAX	65535	unsigned int 类型的最大值
ULONG_MAX	4294967295	unsigned long 的最大值
USHRT_MAX	65535	unsigned short 的最大值

下表所示是<float.h>的一个子集，是与浮点算术运算相关的一些常量。每个值代表相应量的一个最小取值。实际实现时可以定义适当的值。

FLT_RADIX	2	指数表示的基数，如 2、16
FLT_ROUNDS		加法的浮点舍入规则
FLT_DIG	6	float 类型精度(小数位数)
FLT_EPSILON	1E−5	使“1.0 + x != 1.0”成立的最小 x
FLT_MANT_DIG		基数为 FLT_RADIX 的尾数中的数字数
FLT_MAX	1E+37	最大浮点数
FLT_MAX_EXP		使 FLT_RADIXn−1 可表示的最大 n
FLT_MIN	1E−37	最小的规范化浮点数
FLT_MIN_EXP		使 10n 为规范化数的最小 n
DBL_DIG	10	double 类型精度(小数位数)
DBL_EPSILON	1E−9	使“1.0 + x != 1.0”成立的最小 x
DBL_MANT_DIG		基数为 FLT_RADIX 的尾数中的数字数
DBL_MAX	1E+37	最大双精度浮点数
DBL_MAX_EXP		使 FLT_RADIXn−1 可表示的最大 n
DBL_MIN	1E−37	最小的规范化双精度浮点数
DBL_MIN_EXP		使 10n 为规范化数的最小 n

附录Ⅲ　C 语言错误提示

3.1　致命错误信息

1. Bad call of in-line function(内部函数非法调用)

分析:在使用一个宏定义的内部函数时,没能正确调用。一个内部函数以两个下划线“___”开始和结束。

2. Irreducable expression tree(不可约表达式树)

分析：这种错误指文件行中的表达式太复杂，使得代码生成程序无法为它生成代码。这种表达式必须避免使用。

3. Register allocation failure(存储器分配失败)

分析：这种错误指的是文件行中的表达式太复杂，代码生成程序无法为它生成代码。此时应简化这种繁杂的表达式或干脆避免使用它。

3.2　一般错误信息

1. #operator not followed by maco argument name　(#运算符后没跟宏变元名)

分析：在宏定义中，#用于标识一宏变串。#符号后必须跟一个宏变元名。

2. 'xxxxxx' not anargument　('xxxxxx'不是函数参数)

分析：在源程序中将该标识符定义为一个函数参数，但此标识符没有在函数中出现。

3. Ambiguous symbol 'xxxxxx'　(二义性符号'xxxxxx')

分析：两个或多个结构的某一域名相同，但具有的偏移、类型不同。在变量或表达式中引用该域而未带结构名时，会产生二义性，此时需修改某个域名或在引用时加上结构名。

4. Argument # missing name　(参数#名丢失)

分析：参数名已脱离用于定义函数的函数原型。如果函数以原型定义，该函数必须包含所有的参数名。

5. Argument list syntax error　(参数表出现语法错误)

分析：函数调用的参数间必须以逗号隔开，并以一个右括号结束。若源文件中含有一个其后不是逗号也不是右括号的参数，则出错。

6. Array bounds missing　(数组的界限符“]”丢失)

分析：在源文件中定义了一个数组，但此数组没有以方括号“]”结束。

7. Array size too large　(数组太大)

分析：定义的数组太大，超过了可用的内存空间。

8. Assembler statement too long　(汇编语句太长)

分析：内部汇编语句最长不能超过 480 字节。

9. Bad configuration file　(配置文件不正确)

分析：TURBOC.CFG 配置文件中包含了不是合适命令行选择项的非注解文字。配置文件命令选择项必须以一个短横线开始。

10. Bad file name format in include directive　(包含指令中文件名格式不正确)

分析：包含文件名必须用引号（“filename.h”）或尖括号（<filename>）括起来，否则将产生本类错误。如果使用了宏，则产生的扩展文本也不正确，因为无引号没办法识别。

11. Bad ifdef directive syntax （ifdef 指令语法错误）

分析：#ifdef 必须以单个标识符（只此一个）作为该指令的体。

12. Bad ifndef directive syntax （ifndef 指令语法错误）

分析：#ifndef 必须以单个标识符（只此一个）作为该指令的体。

13. Bad undef directive syntax （undef 指令语法错误）

分析：#undef 指令必须以单个标识符（只此一个）作为该指令的体。

14. Bad file size syntax （位字段长语法错误）

分析：一个位字段长必须是 1～16 位的常量表达式。

15. Call of non-functin （调用未定义函数）

分析：正被调用的函数无定义，通常由于不正确的函数声明或函数名拼错而造成。

16. Cannot modify a const object （不能修改一个常量对象）

分析：对定义为常量的对象进行不合法操作（如对其赋值）引起本错误。

17. Case outside of switch （Case 出现在 switch 外）

分析：编译程序发现 Case 语句出现在 switch 语句之外，这类故障通常是由于括号不匹配造成的。

18. Case statement missing （漏掉 Case 语句）

分析：Case 语句必须包含一个以冒号结束的常量表达式，如果漏了冒号或在冒号前多了其它符号会出现此类错误。

19. Character constant too long （字符常量太长）

分析：字符常量的长度通常只能是一个或两个字符长，超过此长度则会出现这种错误。

20. Compound statement missing （漏掉复合语句）

分析：编译程序扫描到源文件末时，未发现结束符号（大括号），此类故障通常是由于大括号不匹配所致。

21. Conflicting type modifiers （类型修饰符冲突）

分析：对同一指针，只能指定一种变址修饰符（如 near 或 far）；而对于同一函数，也只能给出一种语言修饰符（如 Cdecl、pascal 或 interrupt）。

22. Constant expression required （需要常量表达式）

分析与处理：数组的大小必须是常量，本错误通常是由于#define 常量的拼写错误引起。

23. Could not find file 'xxxxxx.xxx' （找不到'xxxxxx.xx'文件）

分析：编译程序找不到命令行上给出的文件。

24. Declaration missing （漏掉了说明）

分析：当源文件中包含了一个 struct 或 union 域声明，而后面漏掉了分号，则会出现此类错误。

25. Declaration needs type or storage class （必须给出类型或存储类）

分析与处理：正确的变量说明必须指出变量类型，否则会出现此类错误。

26. Declaration syntax error （出现语法错误）

分析：在源文件中，若某个说明丢失了某些符号或输入多余的符号，则会出现此类错误。

27. Default outside of switch （Default 语句在 switch 语句外出现）

分析：这类错误通常是由于括号不匹配引起的。

28. Define directive needs an identifier （Define 指令必须有一个标识符）

分析：#define 后面的第一个非空格符必须是一个标识符，若该位置出现其他字符，则会引起此类错误。

29. Division by zero （除数为零）

分析：当源文件的常量表达式出现除数为零的情况，则会造成此类错误。

30. Do statement must have while （do 语句中必须有 while 关键字）

分析：若源文件中包含了一个无 while 关键字的 do 语句，则出现本错误。

31. DO while statement missing （do while 语句中漏掉了符号“(”）

分析：在 do 语句中，若 while 关键字后无左括号，则出现本错误。

32. Do while statement missing ;(do while 语句中漏掉了分号）

分析：在 DO 语句的条件表达式中，若右括号后面无分号则出现此类错误。

33. Duplicate Case （Case 情况不唯一）

分析：Switch 语句的每个 case 必须有一个唯一的常量表达式值，否则导致此类错误发生。

34. Enum syntax error （Enum 语法错误）

分析：若 enum 说明的标识符表格式不对，会引起此类错误发生。

35. Enumeration constant syntax error （枚举常量语法错误）

分析：若赋给 enum 类型变量的表达式值不为常量，则会导致此类错误发生。

36. Error Directive : xxxx （错误指令：xxxx）

分析：源文件处理#error 指令时，显示该指令指出的信息。

37. Error Writing output file （写输出文件错误）

分析：这类错误通常是由于磁盘空间已满，无法进行写入操作而造成的。

38. Expression syntax error （表达式语法错误）

分析：本错误通常是由于出现两个连续的操作符、括号不匹配或缺少括号、前一语句漏掉了分号引起的。

39. Extra parameter in call （调用时出现多余参数）

分析：本错误是调用函数时，其实际参数个数多于函数定义中的参数个数所致。

40. Extra parameter in call to xxxxxx （调用 xxxxxx 函数时出现了多余参数）

41. File name too long （文件名太长）

分析：#include 指令给出的文件名太长，致使编译程序无法处理，则会出现此类错误。通常 DOS 下的文件名长度不能超过 64 个字符。

42. For statement missing ） （for 语句缺少“)”符号）

分析：如果 for 语句中控制表达式后缺少右括号，则会出现此类错误。

43. For statement missing （ （for 语句缺少“(” 符号）

44. For statement missing ; （for 语句缺少“;”符号）

分析：如果 for 语句某个表达式后缺少分号，则会出现此类错误。

45. Function call missing ） （函数调用缺少“)”符号）

分析：如果函数调用的参数表漏掉了右括号或括号不匹配，则会出现此类错误。

46. Function definition out ofplace （函数定义位置错误）

47. Function doesn't take a variable number of argument （函数不接受可变的参数个数）

48. Goto statement missing label （goto 语句缺少标号）

49. If statement missing （ （if 语句缺少“(”符号）

50. If statement missing ）（if 语句缺少“)”符号）
51. Illegal initialization （非法初始化）
52. Illegal octal digit （非法八进制数）

分析：此类错误通常是由于八进制常数中包含了非八进制数字所致。

53. Illegal pointer subtraction （非法指针相减）
54. Illegal structure operation （非法结构操作）
55. Illegal use of floating point （浮点运算非法）
56. Illegal use of pointer （指针使用非法）
57. Improper use of a typedef symbol （typedef 符号使用不当）
58. Incompatible storage class （不相容的存储类型）
59. Incompatible type conversion （不相容的类型转换）
60. Incorrect commadn line argument: xxxxxx （不正确的命令行参数：xxxxxx）
61. Incorrect commadn file argument: xxxxxx （不正确的配置文件参数：xxxxxx）
62. Incorrect number format （不正确的数据格式）
63. Incorrect use of default （不正确使用 deflult）
64. Initializer syntax error （初始化语法错误）
65. Invaild indirection （无效的间接运算）
66. Invalid macro argument separator （无效的宏参数分隔符）
67. Invalid pointer addition （无效的指针相加）
68. Invalid use of dot （点符号使用出错）
69. Macro argument syntax error （宏参数语法错误）
70. Macro expansion too long （宏扩展太长）
71. Mismatch number of parameters in definition （定义中参数个数不匹配）
72. Misplaced break （break 位置出错）
73. Misplaced continue （continue 位置出错）
74. Misplaced decimal point （十进制小数点位置错）
75. Misplaced else （else 位置错）
76. Misplaced else directive （else 指令位置错）
77. Misplaced endif directive （endif 指令位置错）
78. Must be addressable （必须是可编址的）
79. Must take address of memory location （必须是内存一地址）
80. No file name ending （没有文件终止符）
81. No file names given （未给出文件名）
82. Non-portable pointer assignment （对不可移植的指针赋值）
83. Non-portable pointer comparison （不可移植的指针比较）
84. Non-portable return type conversion （不可移植的返回类型转换）
85. Not an allowed type （不允许的类型）
86. Out of memory （内存不够）
87. Pointer required on left side of （操作符左边须是一指针）
88. Redeclaration of 'xxxxxx' （'xxxxxx'重定义）
89. Size of structure or array not known （结构或数组大小不定）

90. Statement missing ; （语句缺少“;”符号）
91. Structure or union syntax error （结构或联合语法错误）
92. Structure size too large （结构太大）
93. Subscription missing ］（下标缺少“]”符号）
94. Switch statement missing （ （switch 语句缺少“(”符号）
95. Switch statement missing ） （switch 语句缺少“)”符号）
96. Too few parameters in call （函数调用参数太少）
97. Too few parameter in call to 'xxxxxx' （调用'xxxxxx'时参数太少）
98. Too many cases （case 太多）
99. Too many decimal points （十进制小数点太多）
100. Too many default cases （defaut 太多）
101. Too many exponents （阶码太多）
102. Too many initializers （初始化太多）
103. Too many storage classes in declaration （说明中存储类太多）
104. Too many types in declaration （说明中类型太多）
105. Too much auto memory in function （函数中自动存储太多）
106. Too much global define in file （文件中定义的全局数据太多）
107. Two consecutive dots （两个连续点）
108. Type mismatch in parameter # （参数#类型不匹配）
109. Type mismatch in parameter # in call to 'XXXXXXX' （调用'XXXXXXX'时参数#类型不匹配）
110. Type missmatch in parameter 'XXXXXXX' （参数'XXXXXXX'类型不匹配）
111. Type mismatch in parameter 'XXXXXX' in call to 'YYYYYYY' （调用'YYYYYYY'时参数'XXXXXX'数型不匹配）
112. Type mismatch in redeclaration of 'XXX' （重定义类型不匹配）
113. Unable to create output file 'XXXXXX.XXX' （不能创建输出文件'XXXXXX.XXX'）
114. Unable to create turboc.lnk （不能创建 turboc.lnk）
115. Unable to execute command 'xxxxxxxx' （不能执行'xxxxxxxx'命令）
116. Unable to open include file 'xxxxxx.xxx' （不能打开包含文件'xxxxxxx.xxx'）
117. Unable to open inputfile 'xxxxxx.xxx' （不能打开输入文件'xxxxxxx.xxx'）
118. Undefined label 'xxxxxxx' （标号'xxxxxxx'未定义）
119. Undefined structure 'xxxxxxxxx' （结构'xxxxxxxxxx'未定义）
120. Undefined symbol 'xxxxxxx' （符号'xxxxxxxx'未定义）
121. Unexpected end of file in comment started on line # （源文件在某个注释中意外结束）
122. Unexpected end of file in conditional stated on line # （源文件在#行开始的条件语句中意外结束）
123. Unknown preprocessor directive 'xxx' （不认识的预处理指令：'xxx'）
124. Untermimated character constant （未终结的字符常量）
125. Unterminated string （未终结的串）
126. Unterminated string or character constant （未终结的串或字符常量）

127. User break （用户中断）
128. Value required （赋值请求）
129. While statement missing（ （While 语句漏掉“（”符号）
130. While statement missing ） （While 语句漏掉“）”符号）
131. Wrong number of arguments in of 'xxxxxxxx' （调用'xxxxxxxx'时参数个数错误）

附录Ⅳ　编 程 风 格

1. 空白

编程绝对不能吝惜空间，在 C 语言的语法中虽然允许多条语句写在同一行中，但作为一个具有良好风格的程序应在一行中书写一条语句。尽量留空白。

例如：

```
for（i=0; i<=100; i++）
    sum = sum + i;

printf("sum=%d\n", sum);
```

在 C 语言语法中也允许一条语句书写在多行，但有的语句是不可以的，如 printf 语句中的格式控制字符串不可分为多行书写，必须写在同一行中。如下面的语句将无法通过编译。

```
printf("Hello
    World!");
```

2. 对齐

类似于留空白的原则，程序代码不要顶格写，一般情况函数内部的语句都不要顶格写，要留出空白，输入代码时，可以用 space 键，也可以用 Tab 键留空白，但很多情况下，特别是在行首的时候采用后者，它可以方便进行对齐。

大括号（“{”、“}”）是 C 语言中经常用到的分隔符，在 C 语言程序中的大括号要占一行，不要和其他程序代码放在同一行。

除表示函数体的大括号外，函数体内部的每一对大括号构成了一条复合语句，复合语句内部的语句应与复合语句外部的语句之间留出缩进，如下面程序段。这种编程风格经常用于 if…else 语句、for 语句、while 语句和 do…while 语句等。

```
1  #include<stdio.h>
2  void  main()
3  {
4      int  grade, cnt1=0, cnt2=0;
5      double  avg=0.0;
6      printf("请输入多名学生的成绩，输入负数退出。\n");
7      scanf("%d", &grade);
8      while(grade>=0)
9      {
10         if(grade>=60)
11             cnt1++;                    // 统计及格人数
12         cnt2++;                        // 统计学生人数
13         avg=avg+grade;
14         scanf("%d", &grade);
```

```
    }
    if(cnt2!=0)
        {
            avg=avg/cnt2;                    // 求平均分
            printf("及格人数%d.\n 平均分%.2f.\n", cnt1, avg);
        }
    else
        printf("学生人数为 0\n");
}
```

3. 注释

编写完成的程序并不是只给自己看，特别在一些大项目中，经常通过团队合作完成一个项目，此时程序的可读性就非常重要，在合适的位置添加注释，可以增加程序的可读性。

应养成在输入程序代码的同时添加注释的习惯，浏览注释可以帮助理解程序。有了注释就可以快速浏览程序并找出需要修改的地方，从而方便程序维护。

不要滥用注释，通常情况下隔几行才添加一条注释，不需要每行都添加注释。

4. 比较运算符“==”的使用

在 C 语言中，“=”符号代表赋值运算符，“=”不是数学中判断“等于”关系的运算符，要把它与关系运算符“==”区分开来。在“==”的使用上应注重一些技巧，

在比较变量和常量是否相等时，通常将常量写在前，变量在后。

例如在比较某个变量和某个常量的值是否相等时，如果使用以下程序段：

```
if(x= =5)
    printf("是");
else
    printf("否");
```

则在输入 if 语句中的表达式时，经常会写成 if(x=5)，在很多情况下，这样的程序的运行结果肯定是错误的，这属于逻辑错误，编译器无法识别。这样的错误很难排除，解决方法是将逻辑错误变为语法错误。例如把上述程序写成：

```
if(5= =x)
    printf("是");
else
    printf("否");
```

对以上程序段，如果不小心把 if(5= =x)写成 if(5=x)，编译时，系统就会报错，从而将错误制止在摇篮中。

附录Ⅴ　全国计算机等级考试二级 C 语言程序设计考试大纲

(2013 年版)

基本要求

1. 熟悉 Visual C++ 6.0 集成开发环境。
2. 掌握结构化程序设计的方法，具有良好的程序设计风格。
3. 掌握程序设计中简单的数据结构和算法并能阅读简单的程序。
4. 在 Visual C++ 6.0 集成环境下，能够编写简单的 C 程序，并具有基本的纠错和调试程序的能力。

考试内容

一、C 语言程序的结构

1. 程序的构成，main 函数和其他函数。
2. 头文件，数据说明，函数的开始和结束标志以及程序中的注释。
3. 源程序的书写格式。
4. C 语言的风格。

二、数据类型及其运算

1. C 的数据类型(基本类型，构造类型，指针类型，无值类型)及其定义方法。
2. C 运算符的种类、运算优先级和结合性。
3. 不同类型数据间的转换与运算。
4. C 表达式类型(赋值表达式，算术表达式，关系表达式，逻辑表达式，条件表达式，逗号表达式)和求值规则。

三、基本语句

1. 表达式语句，空语句，复合语句。
2. 输入输出函数的调用，正确输入数据并正确设计输出格式。

四、选择结构程序设计

1. 用 if 语句实现选择结构。
2. 用 switch 语句实现多分支选择结构。
3. 选择结构的嵌套。

五、循环结构程序设计

1. for 循环结构。
2. while 和 do-while 循环结构。
3. continue 语句和 break 语句。
4. 循环的嵌套。

六、数组的定义和引用

1. 一维数组和二维数组的定义、初始化和数组元素的引用。
2. 字符串与字符数组。

七、函数

1. 库函数的正确调用。

2. 函数的定义方法。

3. 函数的类型和返回值。

4. 形式参数与实在参数，参数值的传递。

5. 函数的正确调用，嵌套调用，递归调用。

6. 局部变量和全局变量。

7. 变量的存储类别(自动，静态，寄存器，外部)，变量的作用域和生存期。

八、编译预处理

1. 宏定义和调用(不带参数的宏，带参数的宏)。

2.“文件包含”处理。

九、指针

1. 地址与指针变量的概念，地址运算符与间址运算符。

2. 一维、二维数组和字符串的地址以及指向变量、数组、字符串、函数、结构体的指针变量的定义。通过指针引用以上各类型数据。

3. 用指针作函数参数。

4. 返回地址值的函数。

5. 指针数组，指向指针的指针。

十、结构体(即“结构”)与共同体(即“联合”)

1. 用 typedef 说明一个新类型。

2. 结构体和共用体类型数据的定义和成员的引用。

3. 通过结构体构成链表，单向链表的建立，结点数据的输出、删除与插入。

十一、位运算

1. 位运算符的含义和使用。

2. 简单的位运算。

十二、文件操作

只要求缓冲文件系统(即高级磁盘 I/O 系统)，对非标准缓冲文件系统(即低级磁盘 I/O 系统)不要求。

1. 文件类型指针(FILE 类型指针)。

2. 文件的打开与关闭(fopen，fclose)。

3. 文件的读写(fputc，fgetc，fputs，fgets，fread，fwrite，fprintf，fscanf 函数的应用)，文件的定位(rewind，fseek 函数的应用)。

考试方式

上机考试, 考试时长 120 分钟,满分 100 分。

1. 题型及分值

单项选择题 40 分(含公共基础知识部分 10 分)、操作题 60 分(包括填空题、改错题及编程题)。

2. 考试环境

Visual C++ 6.0。

参考文献

[1] 谭浩强. C 程序设计(第 4 版). 北京：清华大学出版社，2010.
[2] 谭浩强. C 程序设计(第 4 版)学习辅导. 北京：清华大学出版社，2010.
[3] 何钦铭. C 语言程序设计. 北京：高等教育出版社，2008.
[4] 郝兴伟. C/C++程序设计. 北京：高等教育出版社，2010.
[5] Peter van der Linden. C 专家编程(第 2 版). 北京：人民邮电出版社，2008.
[6] 徐士良. C 语言程序设计教程(第 3 版). 北京：人民邮电出版社，2009.
[7] 梁宏涛. C 语言程序设计与应用. 北京：北京邮电大学出版社，2011.
[8] 颜晖. C 语言程序设计实验指导. 北京：高等教育出版社，2008.
[9] 王新. C 语言课程设计. 北京：清华大学出版社，2009
[10] 崔武子. C 程序设计教程(第 3 版). 北京：清华大学出版社，2012.
[11] 苏小红. C 语言大学实用教程(第 3 版). 北京：电子工业出版社，2012.
[12] Bradley L. Jones. 21 天学通 C 语言(第 6 版). 北京：人民邮电出版社，2003.

鸣　谢

本书在编写过程中，以下同志参与了资料收集、文字录入及校对、图片收集与处理等方面的工作，在此一并表示感谢。

潘崇黎、崔荣博、刘雪锋、赵静、仇利克、刘伟、陈晓君、王艳丽、张永正、何娜、徐伶伶、郭玉芝、张德民、张欣、袁鹏、王朋朋、朱爱春、王学玲、葛苏慧、陈淑芳、王克进、吕志芳、吴伟、张蕾妮、王晓妍、孙月江、王莹莹